Continuous integration of embedded security software

Master Thesis

Marco von Raumer

April 2020

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
and
Arnaud Durand
Software Engineering Group

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Acknowledgements

Special thanks to Arnaud Durand, who not only supported and advised me during the whole work, but also provided me with hardware and a very useful piece of software. Likewise, I had good discussions with Christian Amsüss, who even brought us into contact with the people working on the specifications for the protocols utilized in this project. Also, thank you to everyone else who helped me in some way or another.

Abstract

This report focuses on the protocols Object Security for Constrained RESTful Environments (OSCORE) and Ephemeral Diffie-Hellman Over COSE (EDHOC), which enable a new method for secure communication designed for embedded devices. New libraries, implementing said protocols, are written and tested with help of a Continuous Integration (CI) pipeline. Some performance metrics are gathered and example applications can successfully be run on a standard PC as well as a selected embedded device.

Keywords: EDHOC, OSCORE, CI, Internet of Things (IoT), Security

Table of Contents

1 Introduction

1.1. Motivation and Goals

1.1.1. Motivation

Even back in 2012, the potential of Internet of Things (IoT) devices was known and considered to be an integral part of the future internet [14]. With an estimated 5.8 billion IoT endpoints by 2020 [33], the popularity certainly isn't declining anytime soon. In fact, they will probably spread even further, especially since they facilitate automation and data collection.

However, there exist a number of issues with IoT devices, two of which are security and energy efficiency. Those were already apparent in 2012 and are still present today [14]. Due to the low-power nature of such devices, complex security protocols are often not viable, which results in a vast amount of easily hacked targets [13].

One way to approach this problem is to use protocols specifically designed for the use case, with energy efficiency in mind while still guaranteeing security. The downside thereof is that traditionally popular protocols are well supported and new competitors first have to get accepted and gain traction.

A popular way to interact with sensors connected to the internet is over REST-APIs, which traditionally use HTTP(S) as a transport layer and either JSON or XML to encode the data. As an already well established and optimized alternative, the Constrained Application Protocol (CoAP) and Concise Binary Object Representation (CBOR) can be used instead. CoAP offers the advantage that requests/responses can be mapped to HTTP and vice versa.

Building on those protocols, Object Security for Constrained RESTful Environments (OSCORE) offers an end-to-end encrypted security layer while staying compatible with nodes, such as proxies, only supporting CoAP. However, OSCORE has only been standardized very recently (July 2019) and thus, unsurprisingly, there didn't exist any usable implementations at the start of this project.

Closely related to OSCORE, there exists another new protocol called Ephemeral Diffie-Hellman Over COSE (EDHOC). It is intended to complement OSCORE and extend it with the functionality to allow secure key exchanges. At the time of writing, EDHOC is still in draft form and not yet standardized, but hopefully, this might change soon. A previous student, Urs Gerber, has worked on a proof-of-concept implementation for

EDHOC, but the requirements and algorithms have since changed and therefore no usable implementation existed for this protocol either.

1.1.2. Goals

One goal of this project was to build usable (even if not complete) libraries for both OSCORE and EDHOC in order to enable using those protocols and facilitate the secure use of IoT devices. Special focus is laid upon having a clean Application Programming Interface (API), the libraries should be easy to use. Also, they should be compatible with both standard x86 machines and embedded devices. As a representative of the latter, a *SparkFun ESP32 Thing* was chosen for testing and demonstrations (See Section 5.3).

Another goal was to have a Continuous Integration (CI) pipeline that would regularly and automatically run tests as well as collect some performance metrics about typical operations such as a key exchange with EDHOC or protecting a message with OSCORE.

1.2. Organization

Introduction

The introduction first describes the motivation and goals of this work, then lists the content of each chapter, and ends with an overview of the formatting conventions.

Chapter 1: Theoretical Background

This chapter introduces and shortly explains the terms and technologies important for the project, including a more detailed look at the protocols OSCORE and EDHOC, which were implemented as libraries.

Chapter 2: Related Work

This chapter mentions related work on similar topics and to what extent it influenced the project.

Chapter 3: Tools and Dependencies

Several different tools are used during development and testing. This chapter explains why each of them is useful for the project and what the main benefits or issues are.

Chapter 4: Design and Implementation

This chapter sheds light on some of the decisions behind the API design and continuous integration. Likewise, the embedded device used for the project is presented.

Chapter 5: Evaluation

The evaluation starts with the performance results about the execution time of typical operations and stack usage, then follow the limitations of the current implementation. At the end, there is an outlook into the future and how the work could be improved.

Chapter 6: Conclusion

The final chapter serves as a reflection upon the work on the project as a whole. It is a short summary of what went great and what could have gone better.

Appendix

The appendix lists all abbreviations, acronyms and references used in this report.

1.3. Notations and Conventions

- Formatting conventions:
  - In most cases, abbreviations or acronyms are written in full when they appear the first time, such as Advanced Encryption Standard (AES). All further appearances will only be in the short form, e.g. AES.
  - Function names, properties or other special tokens are formatted like edhoc_init_context.
  - Snippets of code or data formats are presented as such:

```
data_2 = (
   ? C_U : bstr,
   G_Y : bstr,
   C_V : bstr,
)
```

- Figure s, Table s and Listings s are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure i.j.*
- Whenever the word "protect" is used, it is meant as in encrypt and authenticate.

2 Theoretical Background

This chapter gives an insight into what kind of protocols and technologies this project builds upon and shortly explains them. Wherever it makes sense, alternatives are mentioned and compared.

2.1. Representational State Transfer (REST)

The Representational State Transfer (REST) [8] is not a protocol but an architectural style, imposing certain restrictions on how clients can interact with web services.

It describes a uniform interface for how to interact with web resources. As the name REST indicates, whenever a resource is accessed, a representation of its state is transferred. While this state can be modified, all requests themselves are required to be stateless. This means a request must include all information needed to complete the desired operation, it cannot depend upon context from a previous request. One advantage thereof is that servers don't have to store any state information between requests and allows for better scalability.

Although REST was introduced 20 years ago, it is still very much relevant today [25]. It is very popular when it comes to modern Application Programming Interfaces (APIs) for web services. The resource model also works well with Internet of Things (IoT) devices and sensors. However, the traditional protocols and formats used for REST-APIs are less suitable (see the following sections).

2.2. Constrained Application Protocol (CoAP)

The modern web still almost exclusively uses the Hypertext Transfer Protocol (HTTP) as application protocol. While it has been incrementally improved over the years and there exist supplementary protocols (e.g. WebSockets to allow direct bidirectional communication), there is essentially one common standard in use. But because HTTP was not designed with the goal of minimizing message overhead or latency, it is often not the ideal choice for IoT applications. Hence a number of alternative protocols have been created to better suit those needs. The Constrained Application Protocol (CoAP) [24] is one of them and offers a lot of advantages. Despite being standardized only comparatively recently in 2014, it already has large community support.

One major difference lies in the encoding format. While the textual format of HTTP is great for readability, with it comes the downside of more verbosity and unnecessary overhead. CoAP on the other hand uses a very compact, binary format and a fixed header size of only 4 bytes. Additional options, similar to HTTP header fields, can be present before the payload but they are not mandatory. Although these issues with HTTP can be partly mitigated by applying compression algorithms, a protocol designed for brevity has the advantage of not requiring the extra processing, which is essential in the context of embedded devices where power is limited.

Another strong point of CoAP lies in the fact that multiple underlying transport protocols are supported. User Datagram Protocol (UDP) is the default but Transmission Control Protocol (TCP), Stream Control Transmission Protocol (SCTP) or even Short Message Service (SMS) can be used. Meanwhile, HTTP used to run exclusively over TCP until version 3, which is still in the draft phase [3]. The benefits of UDP include smaller packet sizes, but also lower latency since there is no need to establish a connection before transmitting data. Even though UDP does not offer reliable transport, CoAP can deal with it by optionally marking messages as confirmable and thus requesting an acknowledgment.

The CoAP protocol comes with options to support using caching and proxies, both of which can be useful for scalability and efficiency. Similar HTTP headers were not present in the original version and only defined in subsequent iterations.

Finally, CoAP provides good interoperability with existing services by supporting “a limited subset of HTTP functionality”[24]. This allows for easy mapping between simple HTTP/CoAP requests or responses and enables incompatible devices to communicate with each other through cross-protocol proxies.

2.3. Concise Binary Object Representation (CBOR)

The most common data formats used for REST-APIs are eXtensible Markup Language (XML) or increasingly often JavaScript Object Notation (JSON). Both formats use a textual representation of data which allows both machines and humans to interpret the information. However, this is only of minor importance in the context of embedded Machine-to-Machine (M2M) communication, where message length is more critical.

Even though information represented with JSON is usually more compact than with XML, it is still way more verbose than binary formats. As another downside, neither XML nor JSON allows direct embedding of binary strings (e.g. encryption keys). Instead, the offending data is typically converted into an ASCII string by encoding it with Base64, thus using 33% more bytes.

The Concise Binary Object Representation (CBOR) [4] was designed as a simpler, more efficient alternative. It is a binary format somewhat similar to JSON, supporting all the same data types as well as binary strings. That way it is possible for all JSON-encoded data to be mapped into CBOR and back again, which is great for compatibility. A notable objective of CBOR is that a decoder/encoder must be implementable in a few lines of code, using only a small amount of resources. This makes the format both easy to port onto new platforms and ideal for embedded environments.

For an example of how data is represented with CBOR compared to other formats, have a look at Listing 2.1. The difference is not immediately visible, but in this case, the same data represented in CBOR uses less than half as much bytes than XML even with all whitespace removed. In practice, even more could be gained by substituting arrays for maps, at the cost of losing descriptive keys. The result would be only 57 bytes long, another reduction of nearly 50%, which might be worthwhile considering that the format isn't human-readable anyway.

The CBOR standard also defines a so-called diagnostic notation loosely based on JSON. It is purely intended to display CBOR-formatted data in a human-readable way and makes debugging much easier. Binary values are usually written in hexadecimal, surrounded with single quotes and prefixed with h. For example, the two bytes 0xCAFE would become h'CAFE'.

2.3.1. Concise Data Definition Language (CDDL)

The Concise Data Definition Language (CDDL) [2] provides a convention for how to easily and unambiguously describe CBOR data structures. It is extensively used in the specifications for COSE, OSCORE and EDHOC. As an example see Listing 2.2.

The first part defines a map with a single valid entry, devices, of the type Devices. As the name suggests, Devices is an array with objects of type Device. The star (*) prefix, allows zero or more repetitions. Unlike with maps, array elements are not named but identified by their position. Thus the element labels are not present in actual data but just serve as descriptions. That's why the label device is not found in the example data (except in the case of XML, where it is the type name).

The Device object is another map with several allowed elements. All of them are mandatory except the last one, which is prefixed with a question mark, meaning it can appear at most once. Note that a parameter can have multiple valid types as is the case with batteryLevel.

2.4. CBOR Object Signing and Encryption (COSE)

Similar to how the Javascript Object Signing and Encryption (JOSE) workgroup defined several standards on how to represent security-related objects in the JSON format, the CBOR Object Signing and Encryption (COSE) [20] protocol specifies a more compact version intended for constrained devices. The key difference is of course that CBOR is used instead of JSON.

COSE specifies all necessary details for signing, encrypting or authenticating data and also how to derive keys. It includes definitions for all required structures as well as identifier numbers for parameters and cryptographic algorithms, which are published by the Internet Assigned Numbers Authority (IANA)[1]. These definitions are especially useful for negotiating algorithms between two devices.

[1]COSE Registries: `https://www.iana.org/assignments/cose/cose.xhtml`

```xml
<devices>
    <device>
        <name>sensor0</name>
        <batteryLevel>0.5</batteryLevel>
        <key>tNqcYs6+RUzPPzv6cp51+Q==</key>
    </device>
    <device>
        <name>sensor1</name>
        <batteryLevel>1</batteryLevel>
        <key>BSJcIrV2Gk26TAG3y8+daQ==</key>
    </device>
</devices>
```

(a) XML: 225 bytes (252 with whitespace)

```json
{
    "devices": [
        {
            "name": "sensor0",
            "batteryLevel": 0.5,
            "key": "tNqcYs6+RUzPPzv6cp51+Q=="
        },
        {
            "name": "sensor1",
            "batteryLevel": 1,
            "key": "BSJcIrV2Gk26TAG3y8+daQ=="
        }
    ]
}
```

(b) JSON: 153 bytes (201 with whitespace)

```
A1 # map(1)
   67 64657669636573 # text(7) "devices"
   82 # array(2)
      A3 # map(3)
         64 6E616D65 # text(4) "name"
         67 73656E736F7230 # text(7) "sensor0"
         6C 626174746572794C6576656C # text(12) "batteryLevel"
         F9 3800 # primitive(0.5)
         63 6B6579 # text(3) "key"
         50 # bytes(16)
            B4DA9C62CEBE454CCF3F3BFA729E75F9
      A3 # map(3)
         64 6E616D65 # text(4) "name"
         67 73656E736F7231 # text(7) "sensor1"
         6C 626174746572794C6576656C # text(12) "batteryLevel"
         01 # unsigned(1)
         63 6B6579 # text(3) "key"
         50 # bytes(16)
            05225C22B5761A4DBA4C01B7CBCF9D69
```

(c) CBOR: 110 bytes
Note: the data is shown in hexadecimal, whitespace and comments are not part of it

Listing 2.1.: A representation of example data in different formats

```
myObject = {
    devices: Devices
}

Devices = [
    * device: Device
]

Device = {
    name: tstr,
    batteryLevel: uint / float,
    key: bstr,
    ? disabled: bool
}
```

Listing 2.2.: One possible CDDL description of the structure of the example data presented previously, in Listing 2.1.

2.5. Object Security for Constrained RESTful Environments (OSCORE)

Object Security for Constrained RESTful Environments (OSCORE) [23] was designed to enable end-to-end encrypted communication using CoAP in a way that's suitable for constrained devices. Unlike (Datagram) Transport Layer Security ((D)TLS) it does not provide an additional security layer but is an extension for CoAP. Messages protected with OSCORE are valid CoAP messages themselves, which offers some key advantages over the other methods.

When using (D)TLS, the whole CoAP message gets encrypted; both the payload and all options are not readable. However, CoAP defines some options necessary for proxies to relay messages. To access those, proxies need to decrypt packets and in the process thereof they might read or modify the payload. With OSCORE, the situation changes: the payload, as well as sensitive options, are protected, but all proxy-related options are included in the unprotected part. Thus, proxies can operate without unprotecting messages and the content stays encrypted end-to-end. This is both beneficial in terms of increased security but also better efficiency. Additionally, because the protection happens in such a transparent manner, only the two endpoints need to support OSCORE. Any proxies in-between are completely unaffected and can be totally unaware of OSCORE.

Of course these points only apply for pure CoAP proxies. In a case where e.g. CoAP is mapped to HTTP it is inevitable to first unprotect OSCORE messages – unless it only happens for an intermediate hop and the message is later mapped back to CoAP, which is possible, too.

Another benefit of OSCORE is that it reuses and builds upon existing technologies, namely COSE and CBOR. As a result, it is possible to share libraries and keep the additional code size lower.

2.5.1. Protocol details

To establish an OSCORE connection, both parties need to be in possession of some shared information, called common context. It includes algorithms, a secret and salt as well as an optional context identifier. Two different algorithms are required:

- An Authenticated Encryption with Associated Data (AEAD) algorithm, which is used to protect the OSCORE payload containing sensitive options and the original payload. Note that an AEAD does more than just encrypt data: on top of providing confidentiality, the authenticity of said data is also ensured.

  The default and mandatory to implement AEAD is AES-CCM-16-64-128, which was assigned the COSE algorithm number 10. It uses AES in CCM mode (AES-CCM) with a 128-bit key, 64-bit tag, 13-byte nonce.

- A key derivation algorithm, or more precisely a Hash-based Message Authentication Code (HMAC) used as a key derivation function (KDF), giving us a KDF based on HMAC (HKDF). It is used for deriving AEAD keys and common initialization vectors (IVs) from the shared secret and salt. An HKDF operates in two stages: first, a pseudorandom key is extracted from the initial key material and optional salt. Then, during expansion, a key of the desired length is generated.

  The default and mandatory to implement HKDF is based on HMAC-SHA-256, which was assigned the COSE algorithm number 5. Underneath, the Secure Hashing Algorithm (SHA) with a hash size of 256 bits is used.

OSCORE itself does not provide a way to negotiate or share a common context but does define default algorithms which are used in the case that no others are preestablished. Similarly, unless a salt was chosen, it defaults to the empty byte string. But regardless of that, the master secret needs to be known upfront. One possible way to securely establish new values for secret and salt is by completing key exchange according to the EDHOC protocol.

From the common context, each party then derives its own sender/recipient context. To do so, both parties have to know the sender IDs of each other[2], since those correspond to the recipient ID at the other end. Then, the master secret and salt, as well as the IDs, are used as inputs for the HKDF in order to generate sender/recipient keys. Once that's done, both parties end up with partially mirrored security contexts (see Figure 2.1).

Two special parameters are not mirrored and remain mutable after the security contexts are established. The sender sequence number, only present in sender contexts, is incremented for each sent message and serves as an ingredient for AEAD nonces. The replay window, only present in recipient contexts, needs to be updated for each received message to prevent replay attacks.

With the security contexts fully established, the client can then create an OSCORE request. To do so, the original CoAP message is broken up in parts depending on the need for protection. The CoAP header and Token are retained, only the message code gets protected and is therefore replaced with the generic code 0.02, standing for POST. Of course, the original message code is not just discarded but included in the protected part, so it can be restored upon unprotecting. CoAP options are treated according to their class, as defined in the standard. Depending on that, they are either left unprotected

[2]The ID of another party can be retrieved from the OSCORE option of an incoming message, it doesn't necessarily have to be known upfront.

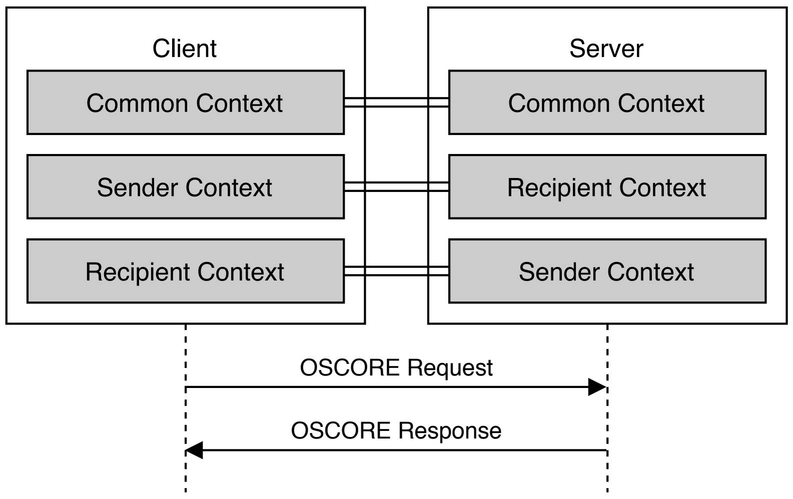


Fig. 2.1.: OSCORE contexts. The = signs indicate how they are partially mirrored. Source: inspired from the illustration in [23].

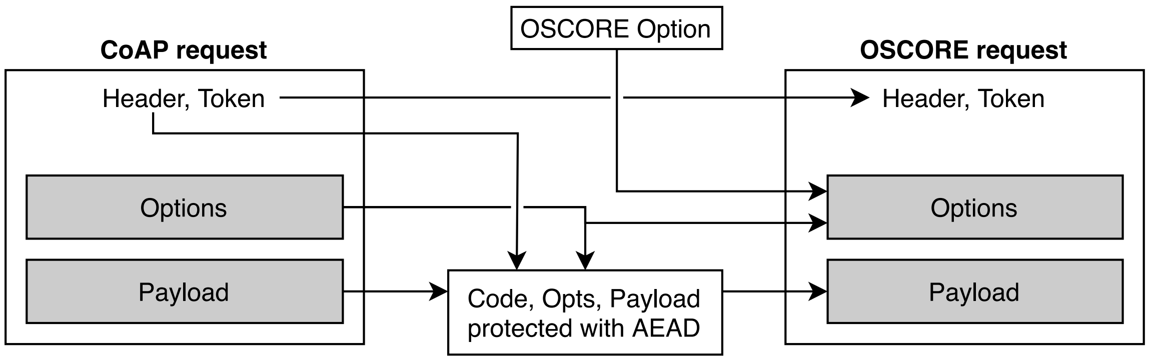


Fig. 2.2.: How an OSCORE request is constructed.

and retained as outer options, or protected and included as inner options. The original payload, if present, is always protected.

Then, the message code, sensitive options and payload are all concatenated and protected using the AEAD algorithm. The resulting ciphertext is used as the new payload. Figure 2.2 illustrates the process and also shows the still missing ingredient: the OSCORE option.

The OSCORE option, assigned the number 9, is a special reserved CoAP option that marks a message as protected with OSCORE. Apart from that, several parameters can be transmitted inside its value. The first byte contains the OSCORE flag bits, indicating which parameters are present, as well as the length of the partial IV. Next follows the partial IV, equivalent to the sender sequence number without any leading zero bytes, used to derive the AEAD nonce. At last, the optional context ID and sender ID finish the sequence. Whether these last two parameters are required, depends on the setup. Context IDs aren't typically necessary unless e.g. multiple machines use the same ID. Additionally, If both parties already know each other's IDs, those don't have to be included either.

Although the OSCORE option has been standardized, some implementations of CoAP libraries might reject it (see Section 6.2), as it was not part of the original CoAP standard.

2.6. Ephemeral Diffie-Hellman Over COSE (EDHOC)

As described in the previous section, OSCORE requires a security context derived from a shared secret and salt. Unless those are preestablished, a key exchange mechanism is needed. The Ephemeral Diffie-Hellman Over COSE (EDHOC) [22] protocol is intended to be used in conjunction with OSCORE and offers exactly this functionality. Please note that as of writing this report, EDHOC is still in a work in progress state and its specifications are not an official standard yet. All the following information about EDHOC is based on draft version 14 (see [22]) and might not be accurate anymore once a standard exists.

Although EDHOC provides a secure way to exchange keys, it cannot perform miracles. To guarantee authenticity, it too requires preestablished keys. At first glance this looks like a vicious, inescapable cycle and the protocol seems rather superfluous. But ultimately, using EDHOC to negotiate shared secrets instead of just relying on a fixed one, does offer two fundamental advantages:

Perfect Forward Secrecy (PFS) Unique keys are generated for each OSCORE session. Thus, if one of these keys gets compromised, no other sessions are affected. On top of that, because EDHOC provides PFS, even if the private authentication key is compromised, no past sessions that were potentially recorded are affected. This is only possible thanks to the additional, ephemeral keys generated to protect one key exchange only.

Asymmetric keys EDHOC allows not only the use of a symmetric shared key but also asymmetric keys or certificates. This can be very convenient since only the public part needs to be distributed and doesn't have to be kept secret.

In fact, this is very similar to how the Transport Layer Security (TLS) handshake works: to establish a trusted connection, certificates are required and new shared keys are negotiated for each session.

Like OSCORE, the protocol uses both COSE and CBOR as well as the same default AEAD algorithm for constructing messages, meaning that some of the functionality can be shared. A major difference though is that EDHOC doesn't have to be used on top of CoAP for transport. It is entirely possible to send raw EDHOC messages over the network, although the specifications do recommend embedding them as payloads in CoAP messages.

2.6.1. Protocol details

The EDHOC key exchange protocol consists of 3 messages which are sent in alternating fashion, or in response to each other, as illustrated by Figure 2.3. It is recommended to send the messages over CoAP, partly because it offers reliable transport. By default, messages should be embedded in the payload of post requests and responses to the Uri-Path "/.well-known/edhoc". Because a client can initiate the protocol either by sending or requesting message 1 (see Figure 2.4), the notion of client/server can easily cause confusion when describing the message structure in the following part. It is much more useful to use the terms party U and V, where party U always sends message 1.

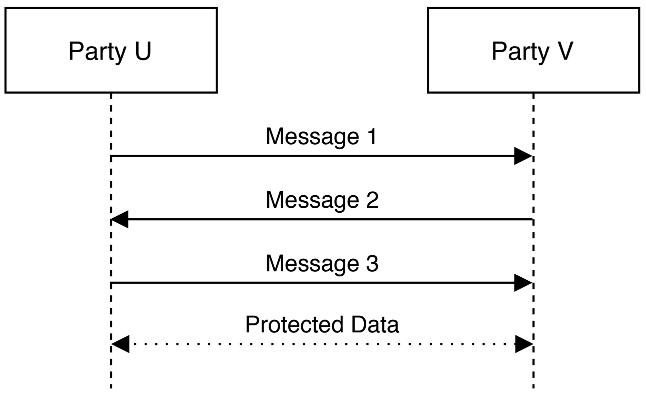


Fig. 2.3.: Successful EDHOC message flow.
Source: inspired from the illustration in [23].

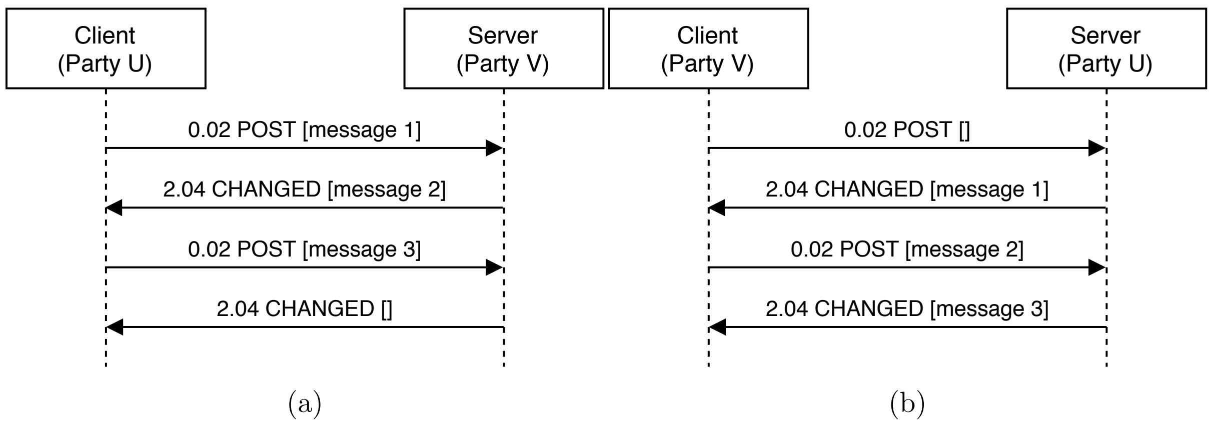


Fig. 2.4.: Normal (a) and alternate (b) EDHOC message flow over CoAP.
The square brackets represent the CoAP payload.
Source: inspired from the illustration in [23].

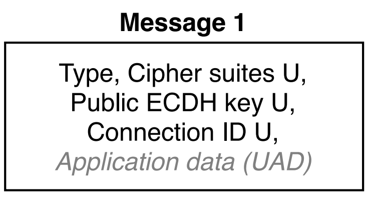


Fig. 2.5.: Structure of EDHOC message 1.

```
message_1 = (
    TYPE : int,
    SUITES_U : suite / [ index : uint, 2* suite ],
    G_X : bstr,
    C_U : bstr,
    ? UAD_1 : bstr,
)

suite = int / [ 4*4 algs: int / tstr, ? para: any ]
```

Listing 2.3.: Specification for EDHOC message 1

EDHOC messages are encoded in CBOR, but instead of an array or map, they consist of CBOR sequences. These are not standardized yet[3], but very simple. Basically, sequences are just zero or more concatenated CBOR objects, or a CBOR array with its header stripped.

As mentioned above, the protocol supports both symmetric and asymmetric authentication, which are distinguished by the "EDHOC Method Type". In both cases, the messages' structure is similar but distinct. For the sake of brevity and also because the implementation only supports asymmetric authentication (see Section 6.2), the symmetric method is not explained in more detail.

Message 1

The key exchange starts with party U sending message 1, the structure of which is illustrated in Figure 2.5. For the specification, see Listing 2.3.

The first element, TYPE can be considered as metadata. It contains information about how the protocol should proceed. More precisely, it is a combination of the EDHOC method, indicating whether symmetric or asymmetric authentication is used, and the correlation parameter. The latter serves to avoid sending IDs whenever responses can be matched by the underlying transport protocol anyway. When using CoAP, the correlation parameter further depends on whether party U is the client or server.

Next, it is required to communicate which cryptographic algorithms should be used. EDHOC needs several different algorithms, the set of which is called a cipher suite. The specifications define one mandatory to implement cipher suite and an alternative, but custom suites could be used as well. A suite consists of an AEAD and HMAC algorithm (for the HKDF), an Elliptic-curve Diffie–Hellman (ECDH) curve, and a signature algo-

[3]CBOR sequences draft: `https://tools.ietf.org/html/draft-ietf-cbor-sequence-02`

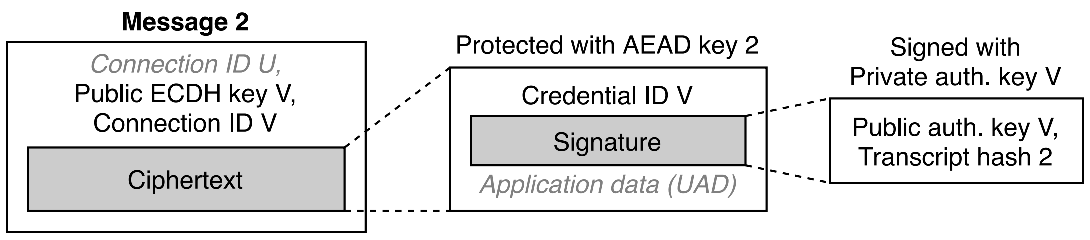


Fig. 2.6.: Structure of EDHOC message 2.

rithm with parameters. What exactly all of them are good for will be explained later on.

The SUITES_U parameter can either be a single chosen cipher suite or an array of supported suites with an index pointing at the selected one. Note that the cipher suite is dictated and not negotiated. The latter exclusively happens if the chosen suite is not supported. Each suite is either an integer label corresponding to a predefined EDHOC cipher suite or an array of COSE algorithm numbers.

In order to provide PFS, EDHOC requires each party to generate an ephemeral ECDH key pair at the beginning of the protocol. Instead of the authentication key, these serve to protect the key exchange. The G_X parameter can be understood[4] as the public key of the ephemeral key pair generated by party U.

Then follows the connection ID of party U, C_U. It serves as a simple identifier that, depending on the correlation parameter, will be included in the response by party V.

At the end of message 1, an optional field can be used to transmit any kind of unprotected application data (UAD).

Message 2

When party V receives message 1, it first checks for the correct structure and whether the selected cipher suite is supported. If anything is wrong, an error message can be sent (see Section 2.6.1). Otherwise, the protocol continues with party U sending message 2. The structure is visualized in Figure 2.6 and defined as presented in Listing 2.4.

Message 2 begins with data_2, which is just another CBOR sequence. The reason why it's grouped separately will be cleared up soon. The first element is the optional connection ID that party U supplied in message 1. It only has to be included depending on the correlation parameter, also received with message 1.

Next follows G_Y the public key of the ephemeral ECDH key pair generated by party V and the connection ID C_V chosen by party V. Again, this can be sent back in message 3 to help match the response.

Then follows the first ciphertext. But before it can be constructed, some more parameters need to be ready. The transcript hash 2 is generated by concatenating the received message 1 with data_2 and applying the hash function of the HMAC algorithm. The pseudorandom key (PRK) is calculated according to the ECDH protocol, by combining party V's private key with party U's public key. By design, party U will be able to

[4]Actually, it is not the public key but only the curve's x coordinate, which is a part of it. But the full key can be derived from it.

```
message_2 = (
    data_2,
    CIPHERTEXT_2 : bstr,
)

data_2 = (
    ? C_U : bstr,
    G_Y : bstr,
    C_V : bstr,
)

plaintext = ( ID_CRED_V / kid_value, signature, ? UAD_2 )
```

Listing 2.4.: Specification for EDHOC message 2

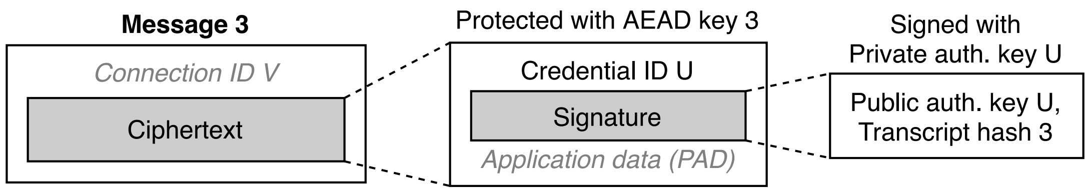


Fig. 2.7.: Structure of EDHOC message 3.

calculate the same PRK from its private and party U's public key. By using the PRK and transcript hash 2 as input to the HKDF algorithm, a shared key (key 2) and nonce are derived, which serve as inputs to the AEAD algorithm that creates the ciphertext.

Said ciphertext is the result of protecting the plaintext consisting of party V's credential ID, signature and optional UAD. The credential ID serves to help party U retrieve party V's public authentication key, which will be needed to verify the signature, which is signed by the counterpart, the private authentication key. The signature is constructed from both the public authentication key and the transcript hash 2. This way, the authenticity is guaranteed since, due to the transcript hash, the signature is only valid for the current message.

Note that even though the UAD is inside the ciphertext, it is considered unprotected because at this point, party U has not authenticated itself yet.

Message 3

Upon receiving message 2, party U will first check the structure, then calculate the PRK, derive the AEAD key 2 and attempt to decrypt the ciphertext. If successful, party V's public authentication key will be retrieved and if found, the signature is verified. If any of these steps fail, an error message can be sent (see Section 2.6.1). Otherwise, party U will proceed to send the message 3, constructed as illustrated in Figure 2.7. Its definition is similar to message 2 and is found in Listing 2.5.

The first part is data_3, which this time only consists of zero or one element. Namely, the connection ID of party V, C_V. As previously explained, it is not necessarily required.

Next follows the ciphertext. To get it, some new parameters are needed. The transcript hash 3 is the result of hashing the concatenation of transcript hash 2, ciphertext from

```
message_3 = (
    data_3,
    CIPHERTEXT_3 : bstr,
)

data_3 = (
    ? C_V : bstr,
)

plaintext = ( ID_CRED_U / kid_value, signature, ? PAD_3 )
```

Listing 2.5.: Specification for EDHOC message 3

message 2 and data_3. Then, by feeding the PRK and the transcript hash 3 to the HKDF algorithm, a new AEAD key (key 3) and nonce are derived. Of course, party U has to calculate the PRK and transcript hash 2 before that, but after having received message 2, all the required information to do so is available.

The ciphertext is the result of protecting the plaintext, containing party U's credential ID, signature and optional protected application data (PAD). Again, the ID serves to facilitate the retrieval of the corresponding public authentication key by the other party. The signature consists of said key and the transcript hash 3, signed by party U's private authentication key.

This time, the optional application data is truly protected since party V already authenticated itself.

Key exchange completion

After party V receives message 3, it is verified in the same manner as message 2 was by party U. Unless there is a problem, both parties can then finish the protocol by deriving the OSCORE master secret and salt or potentially a secret used for another application.

To do so, the transcript hash 4 is required, which is produced by hashing the concatenation of transcript hash 3 and the ciphertext of message 3. Then, by feeding the previously calculated PRK and the transcript hash 4 into the HKDF algorithm, both secret and salt can be derived. If everything goes according to plan, both parties end up with the same values.

Error message

It is entirely possible that two parties don't support the same cipher suites or authentication fails because the required keys are missing. In these cases, or if some other error is detected, the specifications state that an error message, defined as in Listing 2.6, can be sent.

The connection ID, C_x is only included if necessary. Just as with regular messages, this depends on the correlation parameter included in message 1. Following that, an error message describing the problem in a textual, human-readable form. It is supposed to help resolve the problem, but can be empty.

```
error = (
   ? C_x : bstr,
   ERR_MSG : tstr,
   ? SUITES_V : suite / [ 2* suite ],
)
```

Listing 2.6.: Specification for the EDHOC error message

At the end, if the error message is a response to the message 1, the party can list its supported cipher suites such that maybe another one can be agreed upon.

Note that after an error message is sent, the protocol is always aborted and cannot complete anymore. Nevertheless, a new key exchange can be attempted, starting with message 1.

3 Related Work

This chapter shows what related work has already been done by other people and how they influenced the project.

3.1. Authentication and Authorization for Constrained Environments

Urs Gerber has previously done some work in a similar area. In the context of his Master thesis about "Authentication and Authorization for Constrained Environments" [9], he created a proof-of-concept application including an implementation of EDHOC written in C.

But because EDHOC wasn't standardized yet, it kept evolving. Both the message structure and the mandatory algorithms have since changed, causing the implementation to become outdated. Also, the existing code was not ideal for embedded devices. Memory was allocated and freed dynamically, which should be avoided or done in chunks only.

For these reasons, the new EDHOC library was merely inspired by and not based upon the existing implementation. Nevertheless, Gerber's previous work was very helpful to get started and in gaining a deeper understanding of how the protocol actually works. At the time it was, to the knowledge of the author, the only implementation of EDHOC available to tinker with. Even test vectors were not available until the draft got updated in September 2019.

3.2. LibOSCORE

Shortly after the beginning of this project, it was discovered that someone else had started working on an OSCORE library. Christian Amsüss had just about finished the planning phase for libOSCORE and started implementing the first parts. We reached out to him and proposed a collaboration, which he was interested in.

It turned out, that he was in regular contact with members of the group working on the OSCORE and EDHOC definitions and even invited us to an online meeting where we were able to discuss those technologies and eliminate some uncertainties. Not only that, but Amsüss was generally open to questions and also helped us clear up a misunderstanding about how the message correlation mechanism of EDHOC works.

In the end though, the collaboration wasn't as fruitful as hoped. While the goal of this project was to quickly get a usable albeit incomplete library, Amsüss wanted to create a reference implementation, fully reflecting the specification in all its details even if it takes a long time. These incompatible differences in scope and available time frame inevitably lead to us working on another fork and just focusing on getting it to run in the standard case.

3.3. Lightweight Application Layer Protection for Embedded Devices with a Safe Programming Language

Roughly at the same time as this project and under the same supervisor, Martin Disch worked on a very similar topic but with a different target. His focus lay on exploring the suitability of the Rust programming language and ecosystem for embedded devices. To do so, he wrote his own implementation of OSCORE and EDHOC as well as an example application[1].

During the time spent working on our projects, we did share information and attended the meeting with Amsüss together (see above, Section 3.2). But still, our code was written completely independently, mainly because we had different goals and weren't using the same programming language.

[1]OSCORE & EDHOC implementation in Rust: `https://github.com/martindisch/oscore`

4 Tools and Dependencies

This chapter introduces the dependencies and tools used for developing, testing and documenting the libraries and the example applications.

4.1. Dependencies

As the saying goes: "Don't reinvent the wheel". Especially when dealing with security-related topics it makes sense to use well-established libraries instead of writing new and possibly flawed code. For a quick overview, see Figure 4.1.

4.1.1. Libcose

Because both OSCORE and EDHOC build upon COSE, it seemed like a good idea to look for an already existing library providing the necessary functionalities. This is where libcose comes in, as a library designed for constrained devices and aiming to implement the full COSE standard [34]. It allows the use of multiple underlying cryptography libraries, with Sodium (see Section 4.1.2) as the default.

Although this sounded promising, libcose was and to date still remains in an unfinished state, with required algorithms such as AES-CCM and HKDF with HMAC-SHA-256 missing. Therefore, as a temporary solution, a fork[1] was created with those parts implemented as well. Once libcose evolves into a more complete state, it should be possible to switch back to the upstream repository.

[1]Available at: `https://github.com/marcovr/libcose`

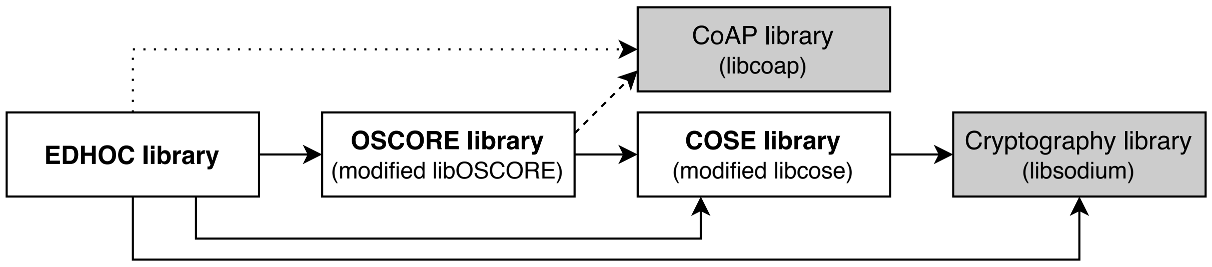


Fig. 4.1.: Dependency graph. Dotted arrows indicate a weaker dependency. Non-modified libraries are marked with a gray background.

Sadly, the library cannot be used to its full potential because OSCORE and EDHOC often take modified COSE structures or even just a single field thereof. Thus, it is sometimes more efficient and makes more sense to call lower-level or internal functions directly.

4.1.2. Sodium

Sodium is a portable library offering easy-to-use and secure functionality not just for encryption but all kinds of tasks involving cryptography. It can be cross-compiled for a variety of platforms and can be used in a lot of common programming languages [36].

Sodium is a fork of the Networking and Cryptography library (NaCl), pronounced “salt”, a library designed to “address the underlying problems”[1] of other implementations by making it simpler to use but still highly performant.

A limitation of Sodium is however, that it only provides a narrow set of algorithms per task. While this may be more than enough for most applications, both OSCORE and EDHOC mandate the use of AES-CCM, which is not supported by Sodium.

4.1.3. TinyCrypt

TinyCrypt is a very compact cryptographic library designed for constrained devices [42]. While it was not used as a dependency per se, it still played an important role during the development phase of this project. Namely, the implementation of AES-CCM, which was added to the custom fork of Libcose, was derived from TinyCrypt.

However, some important changes were made to the code. Specifically, the Advanced Encryption Standard (AES) encryption and decryption functions purely implemented in software were replaced by more efficient hardware instructions, AES New Instructions (AES-NI) for standard x86 machines and the corresponding equivalent for the ESP32 chip.

4.1.4. LibOSCORE

The OSCORE implementation which was later named libOSCORE is not directly a dependency but still a vital part of this project. Originally we planned to work together with its creator Christian Amsüss (see Section 3.2) and to extend libOSCORE with the option for EDHOC key exchange.

But ultimately, because our goals are different and we didn’t have the same expectations, we were forced to use our own fork[2], which was modified so it became actually usable although not offering all features specified in the standard.

Due to the modularity and well-planned structure of libOSCORE, it was easy to reuse functionality for the EDHOC library. For both CoAP and cryptography functions, libOSCORE relies on other libraries but offers general interfaces. This allows the use of different “backends”, by just writing a small wrapper. In fact, the original libOSCORE only includes a mock-implementation for testing purposes as a CoAP library. The support for libcoap was added within our fork.

[2]Available at: `https://gitlab.com/marcovr/oscore-implementation`

Both OSCORE and EDHOC use the CBOR format, but since it is rather simple to implement, no external library is used for it. Instead, libOSCORE was modified such that all CBOR-related operations were extracted into a module and extended to provide more general functionality. Then, these methods are reused by the EDHOC library.

4.1.5. Libcoap

Libcoap is an implementation of the CoAP protocol, with a very extensive set of features. Not only does it allow manipulating and creating CoAP messages, it even includes a fully-fledged server and client as well as (D)TLS support. Of course, examples for how to use all these functionalities were also available, which greatly facilitated the creation of the OSCORE/EDHOC example applications.

In some cases it may be preferable to use a more streamlined alternative, or the operating system might already provide a CoAP library. While libcoap is tightly coupled to the example applications, this is not the case for the OSCORE/EDHOC libraries themselves. There, it is integrated over a generalized interface, so substituting another CoAP library for libcoap should be possible with relatively low effort.

4.2. Build systems

The project is compatible with the following three build systems, serving different purposes each. Depending on the target device, any of them can be used to compile the example applications, but to run the tests, PlatformIO is required.

4.2.1. CMake

The practical tool CMake can be used to easily build the library as well as the example applications for the native platform. Some of the advantages of CMake are its minimal set of dependencies and the integration in various Integrated Development Environments (IDEs). For example, JetBrains' CLion uses it by default. Also, CMake has its own configuration files (CMakeLists.txt) similar to but much simpler and less verbose than standard Makefiles.

4.2.2. ESP-IDF

The Espressif IoT Development Framework (ESP-IDF) is the official development framework for the ESP32 chip [26]. It supplies a toolchain for cross-compiling applications and allows to upload and monitor them on ESP32 devices connected through a serial port interface over USB. At the time of writing, no other chips are supported, but the newly released version 4.0 includes reorganizations as a preparation to do so.

Additionally, ESP-IDF comes with the practical menuconfig tool which allows to easily configure parameters such as log level, WiFi credentials or the address of another machine that the device should communicate with. Figure 4.2 shows how this looks in the case of the EDHOC/OSCORE client example. Due to including the configuration at compile-time and not reading it at runtime, changes are not immediately effective but require a rebuild of the application. Nevertheless, menuconfig is great because it bundles all options together in a well-structured and discoverable way.

The framework provides a modified version of FreeRTOS, a "market-leading real-time operating system (RTOS) for microcontrollers" [31]. A noteworthy change is symmetric multiprocessing (SMP) compatibility [29]: standard FreeRTOS only supports a single core, while the ESP32 processor has up to of them.

Alongside an operating system, ESP-IDF also offers a collection of libraries useful for all kinds of IoT tasks. Namely, libcoap and Sodium were already included, which was very helpful for getting the example applications working on an ESP32 device.

Under its hood, ESP-IDF uses CMake and provides a simplified configuration where dependencies can be specified and only added source files need to be managed. But as a downside thereof, customization options are somewhat limited. Thus, a separate set of configuration files is needed, with changes necessary for this build system.

4.2.3. PlatformIO

Unlike ESP-IDF, PlatformIO is not limited to just the ESP32 chip but can support 34 different platforms and over 700 embedded boards [38]. For this project, it can be used to build the example applications and tests for both the native platform as well as ESP32 devices.

PlatformIO can be used either directly from the command line or from within Visual Studio Code and offers a lot of useful features. For any supported platform, the corresponding toolchains or build systems are collected and set up automatically. The dependency manager takes care of fetching libraries from the PlatformIO library registry or other sources such as GitHub repositories. A unit test system is also included, which allows fully automated building, uploading and testing on connected devices. With a remote test server running, it is even possible to offload the task to another machine connected over the internet.

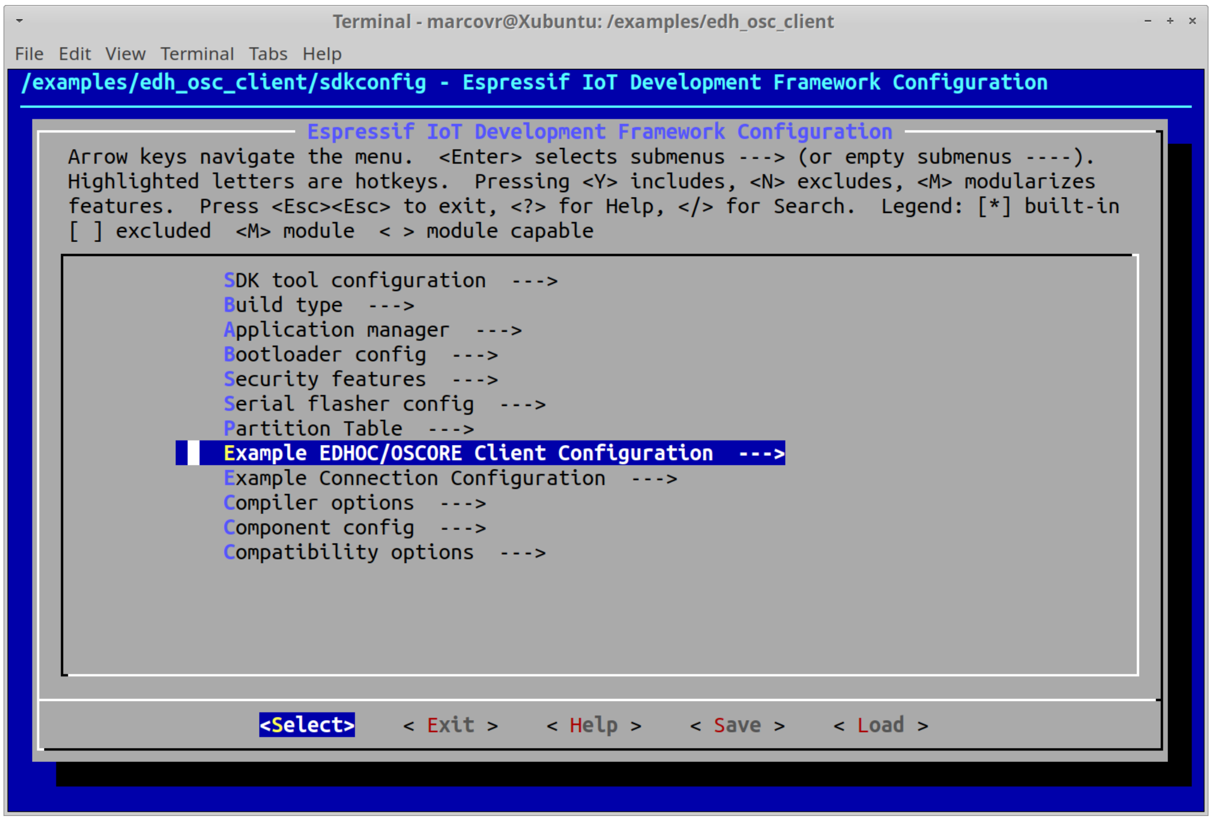
Sadly though, the remote test functionality currently doesn't work for the ESP32 platform because a required configuration file is not transmitted[3]. But once this issue is fixed, that feature would greatly facilitate automated testing with a Continuous Integration (CI) pipeline.

Then there was another problem that we luckily managed to solve. PlatformIO relies on a copy of ESP-IDF to build applications for the ESP32, but it uses its own custom build scripts. While that alone wouldn't have been an issue, it had some interesting consequences. Specifically, the scripts were not yet compatible with version 4.0 of ESP-IDF. However, previous versions only included an older version of libcoap, which didn't provide the necessary functionality for the demo applications.

But thankfully, PlatformIO has a great configuration system which makes it really easy to swap out build components. With the change of a single line in the platformio.ini file, it now fetches a modified variant of the build scripts[4] as well as a slightly patched version of

[3]`https://github.com/platformio/platformio-core/issues/3301`

[4]Modified fork of the Espressif 32 development platform for PlatformIO:
`https://github.com/marcovr/platform-espressif32/tree/feature/idf_v4.0`

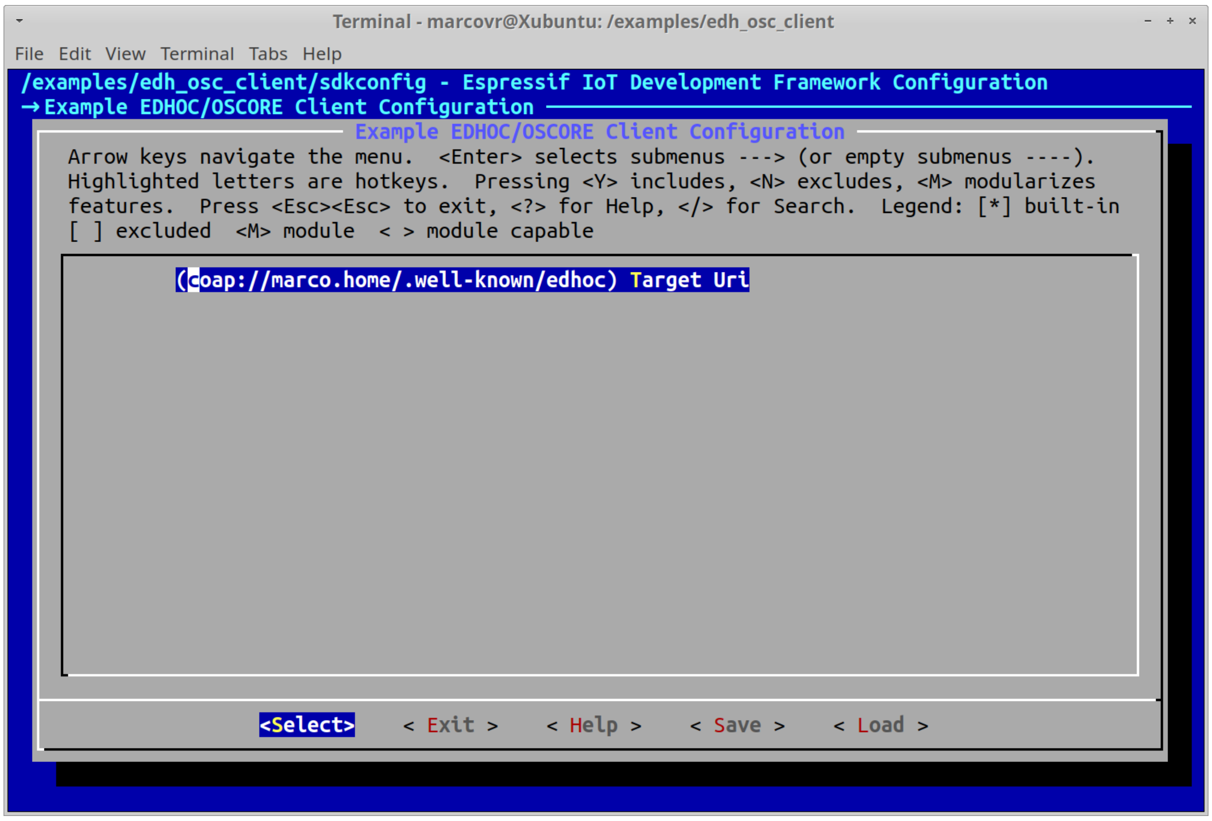

Terminal - marcovr@Xubuntu: /examples/edh_osc_client
File Edit View Terminal Tabs Help

```
/examples/edh_osc_client/sdkconfig - Espressif IoT Development Framework Configuration
                   Espressif IoT Development Framework Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

              SDK tool configuration  --->
              Build type  --->
              Application manager  --->
              Bootloader config  --->
              Security features  --->
              Serial flasher config  --->
              Partition Table  --->
              Example EDHOC/OSCORE Client Configuration  --->
              Example Connection Configuration  --->
              Compiler options  --->
              Component config  --->
              Compatibility options  --->

            <Select>    < Exit >    < Help >    < Save >    < Load >
```

(a) The main menu for the EDHOC/OSCORE client example with a submenu highlighted


Terminal - marcovr@Xubuntu: /examples/edh_osc_client
File Edit View Terminal Tabs Help

```
/examples/edh_osc_client/sdkconfig - Espressif IoT Development Framework Configuration
→ Example EDHOC/OSCORE Client Configuration
                       Example EDHOC/OSCORE Client Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable

         (coap://marco.home/.well-known/edhoc) Target Uri

            <Select>    < Exit >    < Help >    < Save >    < Load >
```

(b) A submenu, where the target URI for the client can be set

Fig. 4.2.: An illustration of how menuconfig can be used to configure options

```c
#include <unity.h>

void example() {
    TEST_ASSERT_TRUE(1);
}

int main() {
    UNITY_BEGIN();
    RUN_TEST(example);
    UNITY_END();
}
```

Listing 4.1.: A minimal demonstration of the Unity Test API.

ESP-IDF[5]. That way, PlatformIO uses the desired version of libcoap and can successfully build the examples.

4.3. Testing

Humans are not perfect, mistakes can always happen. Therefore, any non-trivial code written by humans can and almost certainly will at some point behave differently than expected. Regularly running tests is one way of making sure that errors are found and corrected in time.

But when testing a program or library on a local machine, it can quickly happen that some changes to the code only work because the development environment was configured in a particular way. Then, once the application is deployed onto a new machine, everything breaks because a dependency is missing or data types are of incompatible sizes. This is where CI comes in. A CI pipeline allows to not only test code but also verify that it works on a fresh system and possibly on multiple architectures. All of this is done automatically and the more often the pipeline runs, the sooner any problems are discovered.

4.3.1. Unity Test API

Fortunately, PlatformIO is very well equipped and already includes a unit test system, namely the Unity Test API[6]. It is very simple to use and offers many helpful assertion macros. A minimal example might look like the snippet in Listing 4.1.

Unity automatically aggregates test results and presents them in a neatly arranged and colored overview. Additionally, when used with PlatformIO, it is possible to specify a platform and test filter, in order to run only selected tests on a particular platform.

The test system is not only used to verify the correct behavior of the code, but also to gather performance metrics (see Section 6.1). Ideally, the measured values would be returned by calling a dedicated method for collecting data, however, Unity doesn't have such a functionality. As a substitute, the TEST_IGNORE_MESSAGE macro is called, which prints the supplied text in the output.

[5]Modified fork of ESP-IDF: `https://github.com/marcovr/esp-idf/tree/release/v4.0`

[6]Unity Test API: `https://github.com/ThrowTheSwitch/Unity`, not to be confused with the Unity Test Framework, which is part of the Unity game engine

4.3.2. GitLab CI/CD

"GitLab CI/CD is a powerful tool built into GitLab that allows you to apply all the continuous methods (Continuous Integration, Delivery, and Deployment) to your software with no third-party application or integration needed" [32].

The tool is very well suited for the purpose of automatically testing code in a neutral environment. It offers a huge amount of interesting features, some of which are only available with premium subscriptions, and comprehensive documentation. All of it can easily be managed, mainly by editing the `.gitlab-ci.yml` configuration file.

Thanks to the included Docker support, it is possible to define multiple jobs per repository, each of which will run independently on a specified Docker image. A job can execute arbitrary scripts, build and test applications, deploy a GitLab page (see Section 4.4.2), or produce artifacts.

Every time changes are pushed to the repository, the jobs are executed by so-called "Runners", which are controlled through the coordinator API. It is possible to configure any machine as a Runner just by installing the corresponding software on it and connecting it to GitLab. Alternatively one can profit from shared Runners, virtual machines (VMs) provided by GitLab. With certain limitations, they are completely free.

If the remote test feature of PlatformIO (see Section 4.2.3) was working correctly, it would enable us to deploy onto an actual ESP32 device from such a shared Runner. But although that isn't the case, tests can still be run natively and at least it is possible to verify that builds for the ESP32 platform are successful.

4.3.3. Worst Case Stack Script

The Worst Case Stack script[7] helps to perform static analysis on a C program and generate information about the stack usage. More precisely, it estimates how many bytes of stack memory can maximally be occupied. However, in order to do so, the script needs not just the object files of the program to be analyzed but also some special debug files dumped by the compiler. To get these files, the program to be analyzed must be compiled with gcc and the additional flags `-fdump-rtl-dfinish` and `-fstack-usage`.

But because the debug files are produced during the compilation of translation units, they are not present for linked libraries unless those are compiled with the same options as well. Unfortunately, this is also the case for the standard library, meaning that the stack usage of functions thereof cannot easily be determined.

Thankfully, the script can work around that. Any function calls where the required data is missing, get marked as unbounded but counted as 0 bytes. Thus, an estimation can still be made, with an indication that it's incomplete.

Sadly though, there is another problem: recursive functions. They are detected but without a deeper understanding of the logic behind them, it is impossible to determine how many levels of recursion will occur. Or worse, the maximum depth might not be constant if it depends on the values of received arguments. Thus, an upper bound for stack usage cannot easily be estimated for recursive functions.

[7]Worst Case Stack: `https://github.com/PeterMcKinnis/WorstCaseStack`

For the EDHOC library itself that limitation is not an issue as recursive functions can be avoided. On the other hand, the same can't be said for the dependencies. The Sodium library does use recursion and thus has to be excluded from the estimations.

4.4. Documentation

Thanks to the following two tools, the documentation of this project is both generated and deployed automatically whenever changes are pushed to the repositories.

4.4.1. Doxygen

Doxygen is a great tool for generating documentation. It analyzes the code structure and extracts annotations directly from source files. A lot of commonly used programming languages are supported, including C, and one can choose from multiple available output formats. Particularly interesting is the option to produce Hypertext Markup Language (HTML) files since those can be made available in the form of a website and allow easy navigation with hyperlinks.

By taking the source files as input, Doxygen helps to avoid the typical problem of code changes not being reflected in the documentation. Because annotations are placed right next to the code, it is much harder to forget updating them. Nevertheless, additional text files, even in markdown format, can be included.

Further, Doxygen is very flexible and offers a huge amount of configuration options. For example, the documentation can be extended with interactive graphs depicting file dependencies or function calls.

4.4.2. GitLab Pages

With GitLab Pages, any repository hosted on GitLab can easily be turned into a publicly accessible website. The feature only allows serving static HTML pages or resources and content cannot be dynamically produced with server-side scripts. But in this case that's plenty enough since the documentation does not need to be generated on the fly.

Even better, the static pages don't necessarily have to be part of the repository itself. Instead it is possible to generate them automatically every time the repository gets updated. This works by defining a special job called pages, that GitLab CI/Continuous Delivery/Deployment (CD) will trigger after each push. This job may contain any kind of commands, including Doxygen, which will be executed. The resulting files can then be copied into a configurable output directory.

This is a very elegant way to generate the documentation because it will be always up-to-date and comfortably accessible from the browser, without anyone having to clone the repository and manually running Doxygen.

5 Design and Implementation

5.1. Application Programming Interface (API)

In general, the APIs of both the OSCORE and EDHOC libraries are designed to be easy to use. Only a minimum of initialization is required and all fallible functions have a return code that can be checked against an enumeration with descriptive names and comments. Further, all non-internal methods are documented with explanations for each argument. On top of that, the example applications serve as an illustration for how the APIs can be used to perform an EDHOC key exchange and then share a protected message with OSCORE.

Because the APIs are designed with embedded devices in mind, they don't perform any networking-related operations (see Section 5.3.1) and calls to `malloc`, i.e. dynamic memory allocation is avoided wherever possible. And although some allocations are very hard to get rid of, there is no need to call cleanup functions since all is taken care of.

However, any structures allocated outside need to be cleaned up just as usual. Note that some CoAP libraries such as libcoap do use dynamic allocation, but since the APIs themselves don't create any message objects, this has to be handled outside.

5.1.1. OSCORE

The design for the OSCORE library was mostly left unchanged as it was in libOSCORE. Its API was very well planned and offered almost all essential functionalities. Some extensions were made, either to simplify tasks or to add the few missing parts.

A very nice design choice was the decoupling of dependencies. Because the API includes interfaces for CoAP and cryptography "backends", any suitable library can be integrated just by writing a small wrapper. In fact, this is one of the modifications made: a new wrapper for libcoap was added.

Another interesting idea is that the API allows working directly on unprotected OSCORE messages, without first translating them into proper CoAP messages. Essentially this means that when iterating over CoAP options, the list of them is dynamically zipped together from outer unprotected ones and inner ones, residing inside the formerly encrypted payload. Similarly, when accessing the actual payload, it is located within the outer one. Although this way is very efficient for simply reading the contents, there are some limitations when it comes to writing. For example, existing options cannot be updated unless they are of the exact same size. Indeed most functions for writing to unprotected OSCORE messages are only useful when constructing new messages, before protecting them.

One major change is the addition of the wrap/unwrap methods, e.g. oscore_wrap_request. Previously, it was possible to protect/unprotect OSCORE messages, but there was no convenient way to translate existing CoAP messages into OSCORE messages or the other way around. The wrap/unwrap methods greatly simplify the process by doing both. For example, a normal CoAP message can be passed to oscore_wrap_request and out comes the corresponding protected OSCORE request, ready to be sent.

Some more functionality was missing related to security contexts. The API didn't offer any initialization methods for deriving a security context from a master key and salt. Similarly, there was no way to retrieve a context from a received message. Now both can be done properly.

How to use

Here's a quick rundown of how the API can be used. For a code snippet, see Listing 5.1.

At first, an OSCORE security context has to be initialized, which can be done by calling oscore_context_derive_keys. An OSCORE master context is required as argument, which simply contains the master salt and secret. To obtain these, an EDHOC key exchange can be completed.

Then, depending on what needs to be done, one of the wrap/unwrap methods can be used. For example, to protect a CoAP request, oscore_wrap_request can be called, the result of which can be sent securely. Or to unprotect a received response, oscore_unwrap_response would be used, producing a normal CoAP response.

Where to get

The library and its documentation is available on GitLab:

Source code: `https://gitlab.com/marcovr/oscore-implementation/`

Documentation: `https://marcovr.gitlab.io/oscore-implementation/`

5.1.2. EDHOC

The EDHOC specifications clearly define the messages to be exchanged, but don't limit the choice of the underlying transport protocol. It is however recommended to use CoAP, which seems reasonable considering that EDHOC is intended to be used together with OSCORE, which already uses CoAP anyway.

For that reason, the EDHOC API is designed with the same idea in mind. It offers a set of functions specifically facilitating the key exchange over CoAP alongside a more general set of functions accommodating any other protocol. In fact, every function either reading from or writing to a message comes in two flavors.

The generic functions directly work on raw EDHOC messages, i.e. byte arrays, which can be used for any purpose. Meanwhile, the functions with coap in the name accept CoAP messages in whatever structure the backend defines. This offers the advantage that the

```c
// libOSCORE allows different contexts, making this a little more verbose
struct oscore_context_primitive primitive = {
    .aeadalg = 10,
};
oscore_context_t ctx = {
    .type = OSCORE_CONTEXT_PRIMITIVE,
    .data = (void*)(&primitive)
};

void start_oscore(struct oscore_master_context *master_ctx) {
    oscore_context_derive_keys(&ctx, master_ctx);
}

void send_as_oscore_message(coap_msg *request) {
    coap_msg *protected = create_message();
    oscore_wrap_request(request, protected, &ctx);
    send_message(protected);
}

void on_message_received(coap_msg *response) {
    coap_msg *unprotected = create_message();
    oscore_unwrap_response(response, unprotected, &ctx);
    do_something(unprotected);
}
```

Listing 5.1.: A minimal and simplified example of how to use OSCORE API. No error checking is done and a fictional CoAP library is used.

correct message code is set automatically and optionally even the appropriate Uri-Path and content format options are set. Although the latter two are disabled by default, since a custom Uri-Path might be preferred and the content format encoding is not defined yet, they can be included by setting the EDHOC_COAP_SET_OPTS flag.

How to use

Here's a quick rundown of how the API can be used. For a code snippet, see Listing 5.2.

At first, an EDHOC context needs to be initialized, which holds all the protocol state. This is done by calling edhoc_init_context. The own authentication key pair as well as a list of known public keys of other parties have to be passed as arguments.

The next step depends on whether the protocol should be initiated or a received message has to be answered. In the first case, the method edhoc_start_protocol or one of its CoAP twins[1] has to be called.

If a message is received, edhoc_handle_message or its CoAP twin is the appropriate method to call. Thanks to the EDHOC context, it knows which message to expect and can verify it. Then, it will proceed to generate an appropriate response. This can either be the next message or if something went wrong, e.g. an unsupported cipher suite was chosen, an EDHOC error message. Since the protocol has to be aborted in case of an error, the return code will indicate the failure. However, there are some possible failures when no

[1] There exist two distinct methods for starting the protocol with CoAP, one for each flow direction (see Figure 2.4).

```c
// Keys need to be known beforehand
edhoc_context_t ctx;
edhoc_auth_keypair_t keypair = {...};
size_t peer_keys_length = ...;
edhoc_auth_keypair_t *peer_keys[peer_keys_length] = {...};

void start_edhoc() {
    edhoc_init_context(&ctx, &keypair, peer_keys, peer_keys_length);
    coap_msg *request = create_message();
    edhoc_start_protocol_coap(&ctx, request);
    send_message(request);
}

void on_message_received(coap_msg *response) {
    if (!edhoc_protocol_finished(&ctx)) {
        coap_msg *request = create_message();
        edhoc_handle_coap_message(&ctx, response, request);
        send_message(request);
    } else {
        struct oscore_master_context master_ctx = {
            .master_secret_len = 16,
            .master_salt_len = 8
        };
        edhoc_derive_oscore_params(&ctx, &master_ctx);
        start_oscore(&master_ctx)
        // Now, communication with OSCORE can be started
    }
}
```

Listing 5.2.: A minimal and simplified example of how to use EDHOC API. No error checking is done and a fictional CoAP library is used.

error message is produced, such as if the response buffer is too small. That's why before the response is sent to the other party, it should be checked whether an error message was produced, with the help of the function edhoc_error_message_produced.

As long as the protocol continues without any problems, it will complete after EDHOC message 3 has been sent/received. This can be checked by calling the function edhoc_protocol_finished. After that, it is possible to derive the OSCORE master secret and salt with the help of edhoc_derive_oscore_params. Additional parameters or secrets can be derived by using the EDHOC exporter interface, edhoc_exporter.

Depending on the transport protocol and how many key exchanges are performed at once, it might be necessary to match a received message to an EDHOC context. This can be done by calling edhoc_find_context or its CoAP twin, which will retrieve the corresponding context of a message out of a list of candidates.

Also, a received message might be an EDHOC error message. In that case, the error description in text form (if any) can be extracted with the method edhoc_get_err_msg_reason.

Note that even though transcript hashes are constructed from parts of previous messages (see Section 2.6.1), EDHOC messages don't have to be kept in memory after they are sent/handled. This is achieved by partially creating the hashes and storing them in the EDHOC context.

Where to get

The library and its documentation is available on GitLab:
Source code: `https://gitlab.com/marcovr/edhoc/`
Documentation: `https://marcovr.gitlab.io/edhoc/`

5.2. Continuos integration

5.2.1. Tests

GitLab not only allows to manage Git-repositories but conveniently also offers the possibility to run a CI/CD pipeline for free on shared VMs (see Section 4.3.2). The service is utilized because it allows hassle-free testing, even for other developers creating their own forks later on.

The downside of that approach is of course the lack of embedded devices connected to said VMs. As a result, in the current configuration, tests are only run on the native architecture. But when working with a local machine, this can easily be changed by just replacing the desired target platform. Or, if no platform is specified at all, PlatformIO will run the tests on both the native environment and a connected ESP32 (see Section 5.3). Originally, it was planned to make use of PlatformIO's remote test agent, which would allow to run tests on devices connected over the internet, but attempts to do so were fruitless (see Section 4.2.3).

However, even if no embedded device is available, it can at least be checked whether builds are successful. This is especially important since the EDHOC library can be compiled with three separate build systems. As a result, it happened several times that changes to the code somehow broke one of them, e.g. because a new dependency was not found. With the help of CI such problems can easily be discovered. After setting up a blank environment, installing the required tools and libraries, it is checked that the code successfully builds for both platforms. If anything goes wrong, the job fails and GitLab automatically sends a message containing the last output as well as some additional details.

On the GitLab VMs, native builds finish within a rather short time (less than 30 seconds), thus it can quickly be checked whether both CMake and PlatformIO succeed. On the other hand, building for the ESP32 takes much longer (more than 10 minutes). Because the VMs are only available for free for a limited time, it was decided to skip building with ESP-IDF and to just use PlatformIO. Ideally, ESP-IDF should also be checked, but more often than not, it was PlatformIO causing problems, in particular, due it not yet supporting the desired libraries (see Section 4.2.3).

Although a build may complete without issues, it might not work as expected due to some mistake in the code logic. Thankfully, the specifications for OSCORE and EDHOC include test vectors. By implementing those and running them as tests, it can be verified that the protocol actually does what it's supposed to do.

Sadly though, test vectors don't cover all eventualities. They cannot ensure everything works correctly, some special cases might be handled wrong. An extensive set of unit tests could potentially mitigate the problem, but those take a lot of time to do right. As a compromise, some critical pieces of code, such as the newly implemented HKDF (see Section 4.1.1) are tested more thoroughly.

5.2.2. Other purposes

In addition to just testing, the CI pipeline is also used to collect performance metrics. Of course, when running on the shared GitLab VMs, no data about embedded devices is collected by default. But when running locally, this can easily be changed. For more details about the metrics and results, have a look at Section 6.1.

Further, it was desirable to have an always up-to-date version of the libraries' documentation available online. Again, this is made possible with a CI/CD pipeline (see Section 4.4.2).

5.3. Embedded device

In order to test the libraries and gather performance metrics on an actual embedded device, a suitable board was needed. The following two were evaluated:

Nordic Thingy:91
: A prototyping platform by Nordic Semiconductor with a lot of sensors and an nRF9160 System in Package (SiP). The chip itself includes low power cellular connectivity and GPS positioning while an additional microcontroller provides support for Bluetooth Low Energy (BLE) and Near Field Communication (NFC) [41].

SparkFun ESP32 Thing
: A very compact development platform by SparkFun equipped with an Espressif ESP32 microcontroller, offering integrated WiFi and BLE connectivity. The board has 28 General-Purpose Input/Output (GPIO) pins and can be programmed over USB [27]. It is depicted in Figure 5.1.

In the end, the SparkFun ESP32 Thing was chosen because it was much more convenient. Its WiFi support makes it very easy to establish a connection and being able to program it over USB is compelling. Although the cellular connectivity and GPS of the Nordic Thingy:91 can certainly be very useful for mobility, it is not needed for testing our libraries.

The ESP32 has an "Xtensa®Dual-Core 32-bit LX6 microprocessor" with 520 KByte static random-access memory (SRAM) [5]. This makes the chip a bit more powerful and as a result more power-hungry than other IoT devices, yet it was selected as a representative. It also features hardware acceleration for multiple cryptographic standards, including AES and SHA-2.

5.3.1. Portability

Ideally, the libraries should work not just on the ESP32 but on any embedded device. Platform-specific functions are avoided wherever possible and all networking is left to the application using the libraries. In practice though, some work will be required to get them working on a new platform, although it shouldn't require too much effort.

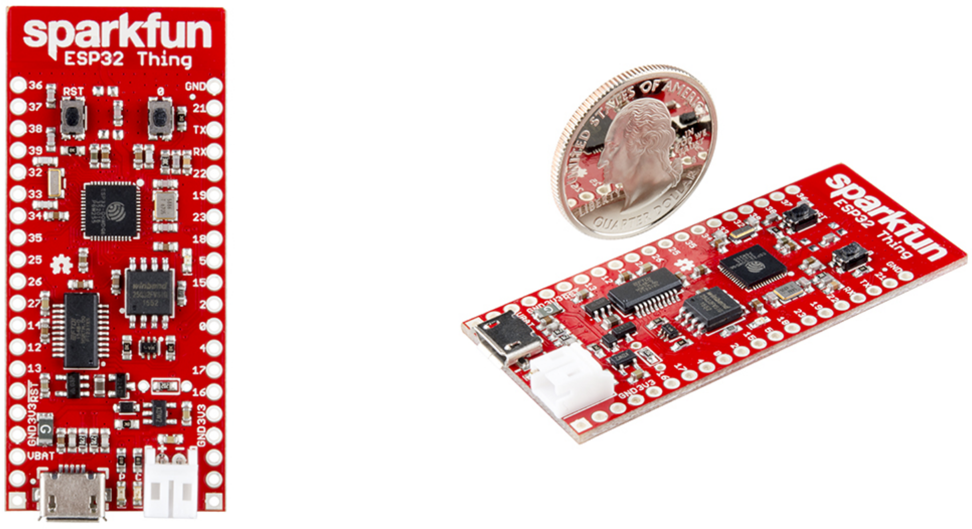


Fig. 5.1.: SparkFun ESP32 Thing.
Source: SparkFun, licensed under CC BY 2.0

One part that definitely requires modifications is our adapted variant of libcose. Because it uses hardware acceleration for AES encryption/decryption, it will certainly not work with any device. A wrapper will have to be written around the platform-specific hardware acceleration functions or, if none are present, a software implementation will be necessary.

Another candidate is the CoAP library. While we are fortunate that libcoap is available for the ESP32, it might not be compatible with other platforms. In that case, it would have to be replaced and a new CoAP backend wrapper would have to be written.

But apart from that, no modifications were needed in order to get the libraries themselves running on the ESP32. Only for the example applications and tests, some minor changes were necessary since the program entry point does some additional setup and requires additional header files.

6 Evaluation

6.1. Performance

To evaluate the newly built libraries, we wanted to measure their performance. How much overhead is there when protecting messages with OSCORE compared to just sending them in clear? How much time does a Key exchange with EDHOC take? How much memory is required to do so?

6.1.1. Metrics

Execution time

A pretty important metric would probably be the energy use, but unfortunately this requires additional hardware and complicates the test setup. Thus, we settled for measuring time, which is not only much easier to do but should also correlate with energy use.

Because a single operation such as protecting a CoAP message takes only a fraction of a millisecond, its duration cannot be measured very precisely. Additionally, on a non-real-time operating system, the results can be tainted by competing processes or other background activity. To solve both problems at once, a task is not run just once, but several times in a row. That way, the total execution time can be divided by the number of completed runs, producing more accurate and average values.

Another issue is network latency, which can vary a lot depending on congestion and interference on the medium. To avoid that altogether, the key exchange is performed entirely on a single device, nothing is sent over the network. Although this means the measured numbers represent ideal conditions that couldn't ever be reached in practice, external influences are eliminated and only the performance of the library matters.

Stack usage

A typical OSCORE operation involves just a single encryption/decryption step as well as reading and writing options one at a time. The bulk amount of memory used depends on the message length and how the chosen CoAP library stores the payload and options. Thus, in this case it doesn't make much sense to measure stack usage.

But this is different with EDHOC. While the message length is not fixed, it doesn't vary too much as long as no extra application data is sent alongside, which the library doesn't currently support anyway. Both for constructing and verifying a message, multiple components and cryptographic operations are required. Because of the very limited amount of stack memory available in embedded devices, this could potentially be problematic. So, analyzing the stack usage during an EDHOC key exchange is much more interesting.

The choice for the kind of metric to measure was easy. The average or minimal stack usage isn't very meaningful, the only relevant value here is the maximal or peak usage in bytes: either the available memory can cope with it or not. The remaining question was, where to get this information? Multiple approaches were considered for how to do so.

At first, let's have a look at the ESP-IDF documentation. It does list functions related to memory management[1] and how much is still available, but that only concerns the heap memory. And even if there was a way to find out how much bytes of stack memory is used at a given time, this would not be very useful since in order to detect peak values, the function would have to be called very frequently.

But of course, there are other ways to reach the goal. Here's a rather crude, experimental one: because the maximum stack size can be set as a parameter when creating a task, one possible course to find out the peak usage is by trial and error. A task can be created and run repeatedly while reducing the available stack size until a stack overflow is encountered. At that point, it is evident that more memory was required.

This method is clearly not an ideal solution. The main reason thereof being that a program cannot recover from a stack overflow. In fact, the documentation states that the "execution of the program can not be continued in a well-defined way" [28]. Thus, if this method was used as part of a test, it would certainly not complete successfully and could cause unexpected side effects. The other problem is that a program might behave differently depending on the input it gets. Sometimes it may require much less memory, resulting in misleading measured values. Luckily, this is not the case when testing a scenario with fixed inputs, such as the test vectors supplied by the specifications.

A much more elegant way to empirically determine the maximal stack usage was only discovered later while preparing this report. It turns out that FreeRTOS, the operating system included in ESP-IDF, provides a function meant exactly for this purpose. It is sufficient to call uxTaskGetStackHighWaterMark once just before the completion of a task, to get the exact number of bytes from the stack that was left untouched during execution [29]. Then, by simply subtracting this number from the maximum amount set at the beginning, the desired value can be calculated.

A totally different approach is to statically analyze the program and deterministically find the maximum stack usage. This way has the big advantage of not relying on any platform-specific features such as the methods described above. Also, it does not require to run any code on an actual embedded device. Therefore, the analysis can be easily completed as part of a CI pipeline, even without a connected ESP32 device.

That being said, finding a suitable tool for the job was much more difficult. Several options were tried and some of them didn't work at all or just returned bogus results, even after a lot of testing. In the end, the Worst Case Stack script is the best one we found. Although it isn't perfect, it works with the standard gcc compiler and does

[1]ESP-IDF, Heap Memory Allocation: `https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/mem_alloc.html`

produce estimates that seem fairly realistic. Unfortunately, because it isn't possible to include all dependencies in the calculations (see Section 4.3.3), the results cannot be directly compared with the experimentally found ones.

Heap usage

The only place where the OSCORE and EDHOC libraries dynamically allocate memory is in the wrapper around libcose. This can be considered a temporary solution because libcose currently requires the additional authenticated data (AAD) for an AEAD algorithm as a single chunk, but it is planned to switch towards an incremental approach[2].

Meanwhile, the API for OSCORE's cryptography backends already allows feeding the AAD in small chunks. Obviously, the wrapper needs to buffer these chunks somewhere, which is done with the help of heap memory since the AAD size is not of fixed size.

Measuring the heap usage is fairly straightforward but there are some caveats. ESP-IDF provides the function `heap_caps_get_minimum_free_size`, which returns the minimal number in bytes of heap memory that were unused up until this point. Then, by subtracting this number from the total amount of available memory, the maximum usage can be calculated. It is important to note that this value also includes all memory used by the system, not just of a single task. Therefore, to isolate the libraries' usage, we need to subtract the number measured by running a dummy task, not allocating any memory.

At first, the results from this approach didn't make any sense. According to them, the EDHOC demo didn't use any heap memory at all, which definitely isn't the case. It turned out that during initialization, the system uses more dynamic memory and when running the EDHOC demo, no new peak is reached. The solution is to just allocate a large chunk of memory at the start of both the demo and the dummy task, causing both of them to reach new peaks of heap memory usage.

6.1.2. Results

Execution time

First, it is important to note that these results were gathered by running the performance tests on a local machine, not on a shared VM provided by GitLab. The values were measured on both an x86-64 machine and an ESP32 device. More precisely, a desktop PC equipped with an Intel Core i5-7600K processor and a SparkFun ESP32 Thing (see Section 5.1).

Of course, as Table 6.1 shows, the time required to complete an EDHOC key exchange is multiple orders of magnitude greater than to (un)protect unprotect an OSCORE message. There are several reasons for that. First, (un)protecting requires just a single encryption/decryption, while a full key exchange involves creating and verifying three separate messages, two of which contain encrypted data. This alone means four encryptions/decryptions already. On top of that, EDHOC uses other cryptographic algorithms as well. Signatures, hashes and key derivation are necessary, but none of these operations profit of hardware acceleration. But since a key exchange is not necessarily required

[2]See this comment on the GitHub issue about adding support for AES-CCM:
`https://github.com/bergzand/libcose/issues/89#issuecomment-518774799`

EDHOC key exchange	**PC (ms)**	**ESP32 (ms)**
only party u	0.289	134
only party v	0.286	133
full exchange	0.572	264

OSCORE operation	**PC (ms)**	**ESP32 (ms)**
protect	0.0024	0.27
protect with payload	0.0072	1.06
unprotect	0.0027	0.24
unprotect with payload	0.0072	1.06

Tab. 6.1.: Measured execution times for EDHOC key exchange and OSCORE operations

for every single connection, they can be done infrequently such that the higher cost is relativized.

Another obvious difference can be observed between the execution times on the PC and the ESP32. Anything else would have been very astonishing, considering that the desktop PC is both vastly more powerful and power-hungry than the embedded device.

The duration of an EDHOC key exchange is not only measured altogether but also broken down into the parts of each party. Unsurprisingly, because both parties have equally complex tasks to do, the parts are finished in almost the same time.

To illustrate the impact of the payload upon the time an OSCORE operation takes to complete, two separate messages are used for testing. Both of them have exactly the same options, taken from the message in the test vectors: Uri-Host: localhost and Uri-Path: tv1. One message is processed without any payload, the other one has an additional 500 bytes long filler text embedded. The difference is clearly visible in the table, on the ESP32, the payload causes an increased duration by roughly a factor of four, while on the PC it's closer to a factor of 3. Note that the measured time includes the creation, (un)protection and verification of a message.

The measured values for execution times of OSCORE operations can be understood as the overhead caused by using OSCORE instead of unprotected CoAP. A comparison of this overhead with another protocol like Datagram Transport Layer Security (DTLS) would be nice to have, but it turned out to be more difficult than expected. Although the ESP-IDF framework includes the *mbed TLS* library which supports both TLS and DTLS, the possibility to just protect a message does not exist. The library takes over networking and just provides methods to send or receive messages. Also, (D)TLS requires a key exchange in the form of a handshake at the beginning of each session, which further complicates a fair comparison.

Stack usage

Again, note that these results were gathered locally, not on a shared VM provided by GitLab. Although, in this case the estimated value by the Worst Case Stack script is exactly the same whether it is calculated locally or in the cloud. The other values were measured directly on an ESP32 device because to get them, certain FreeRTOS functions were (ab)used (see Section 6.1.1). Also, the stack usage typically isn't critical for standard PCs with multiple gigabytes of memory available.

Stack analysis method	**Maximum stack usage (bytes)**
Stack overflow	5981
Watermark	6048
WCS estimation	3624

Tab. 6.2.: Measured maximum stack usage determined with different methods

The values listed in Table 6.2 indicate the estimated and measured maximum stack usage in bytes during a full EDHOC key exchange, as described in Section 6.1.1. Note that no networking is involved, the exchange is performed on one machine only, having access to the data of both parties. Further, the EDHOC messages are not embedded as a CoAP payload, just the raw messages are used.

For this purpose, the EDHOC demo, from edhoc-demo.c, is used where everything is kept in the stack. This includes authentication keys, ephemeral keys, all message buffers and expected messages to compare with. Altogether, this data sums up to just under 900 bytes, which could potentially be stored somewhere else.

As predicted, the estimation done with the Worst Case Stack script is much lower than the measured values. This is due to the fact that not all dependencies are included in the calculation, as explained in Section 6.1.1. Another explanation might be that the estimation is simply inaccurate. Potentially, the ESP32 manages the stack differently, even with the stack smashing protection mode disabled.

It is unclear why the two experimentally gathered values aren't matching. While they are certainly very close, there is an inexplicable difference of 67 bytes. It seems rather unlikely that this would be a safety margin included in the watermark.

Heap usage

As already mentioned in Section 6.1.1, the libraries only use a small amount of heap memory. In fact, during the EDHOC demo, only 52 bytes are allocated at once. But note that this excludes the usage caused by CoAP libraries potentially allocating memory for the messages, such as libcoap does.

With the OSCORE test vectors, even less dynamically allocated memory is used. Only 20 bytes were measured, again without taking the CoAP library into account. In practice this value could be higher though, since the AAD includes the sender/recipient ID, which is of variable length. Theoretically, even CoAP options could be included in the AAD, in order to guarantee their integrity, while not encrypting them. But currently, no such options are defined [23] and the OSCORE library does not support this.

6.2. Limitations

Both the EDHOC and OSCORE libraries do not fully implement the specifications but are limited to basic use-cases. Similarly, the CI pipeline has some shortcomings. The following is a list of the current limitations.

EDHOC

- The EDHOC library follows the specification of draft version 14 ([22]), but since the protocol isn't standardized yet, some details could still change.
- The specification describes two separate methods of authentication and only one of them is available. For now, only the method with asymmetric keys is possible, symmetric keys cannot be used. The main reason for it is that for starters, one method was considered to be enough to get a usable library. Support for both methods would be nice, but is not essential. Also, it would require different processing logic for the messages.
- As a further restriction to the point above, only raw public keys can be used. The EDHOC draft also specifies X.509[3] certificates as an option for the asymmetric method.
- While the EDHOC protocol allows optional UAD as part of the messages 1 and 2 and optional PAD as part of message 3, this is currently not supported. Again, this was not deemed to be an essential feature, since data can just as well be sent either unprotected before the key exchange or afterward, protected with e.g. OSCORE.
- The EDHOC protocol defines a way to negotiate which set of cryptographic algorithms (cipher suite) to use. However, at the moment only the mandatory cipher suite is accepted as no others are supported. Some effort was made towards adding alternative algorithms, but this remains unfinished.

OSCORE

- The OSCORE library follows the specifications from the standard ([23]), but does not implement any of the optional features. Thus, the CoAP options for *observe*, *block-wise transfer*, and *no-response* are not supported. As they are not mandatory anyway, they weren't considered as a priority.
- In addition to the previous point, some of the standard CoAP options have to be treated specially, which isn't currently done. As an example, the Proxy-Uri should be decomposed in separate options and default values for Uri-Host and Uri-Port should be omitted.
- When using libcoap as a CoAP backend, the correct OSCORE option number as defined in the standard cannot be used because messages containing it are detected as malformed and rejected. As a temporary solution to this, another option number, such as the one for the ETag option, can be misused as a replacement. This is obviously not at all ideal and will only work if both endpoints agree on it. Alternatively, a different CoAP library could be used, or libcoap could be modified to accept the OSCORE option.
- The standard defines a default set of algorithms to use for encryption and key derivation. Although these are fully supported, no alternatives are available.
- With OSCORE, CoAP options can either be encrypted (inner options) or not (outer options). The standard further distinguishes between two types of outer options:

[3]X.509 is an International Telecommunications Union (ITU) standard for public key certificates used e.g. for TLS.

normal unprotected ones and special ones which are checked for integrity. The latter type is not supported, but currently no such options are defined anyway.

Continuous integration

- The CI for the OSCORE library itself does only use CMake: tests are only built and run on the native architecture. This isn't too much of an issue though since it is also built together with the EDHOC library.
- With both libraries, it is not checked whether the ESP-IDF build system runs successfully (see Section 5.2.1).
- There are only very few tests, mainly for the test vectors from the specifications. But these aren't very thorough and don't cover special cases.
- By default, no tests are run on an embedded device but just on the native architecture.

6.3. Future Work

As mentioned before, the specifications for EDHOC are not standardized yet and thus even major changes cannot be ruled out. In fact, the draft has already been updated once in the time between finishing the implementation and writing this report. Luckily though, the additions do not seem to affect the existing library.

But still, once the protocol is properly standardized, it might be necessary to adapt the code. Also, it would certainly be beneficial to fully implement the standard, as there is some missing functionality (see Section 6.2).

Similarly, once libOSCORE is in a more mature state, it should be considered as a replacement for the current OSCORE library. If its development does not unexpectedly come to a stop, then without a doubt it will at some point become usable and provide a lot more features.

Concerning the CI, it would definitely be nice to have more unit tests. Currently, it is verified that the library can be built and test vectors pass, but this doesn't cover special cases. The correctness of individual functions is only partially tested, which means mistakes could happen.

7 Conclusion

7.1. Review

As with most projects, there were some ups and downs. Despite not working out of the box, PlatformIO ended up being very useful as it allows to easily manage multiple different platforms, to build, deploy, and run tests on them. But other things didn't quite go as expected. Originally, it was planned to incorporate the *ATECC608A secure element* into the project. The idea was to add support for it in the libraries, such that an embedded device could offload cryptographic operations and the element would take care of them. But as it turned out, the updated EDHOC specifications changed the mandatory-to-implement algorithms and as a result the element was incompatible. Therefore, the idea had to be abandoned.

Although not everything went according to plan, overall the project has been a success. Both the OSCORE and EDHOC libraries are definitely usable and all changes automatically go through the CI pipeline. The documentation and all the source code, including CI configurations and tests, is publicly accessible:

OSCORE
Source code: `https://gitlab.com/marcovr/oscore-implementation/`
Documentation: `https://marcovr.gitlab.io/oscore-implementation/`

EDHOC
Source code: `https://gitlab.com/marcovr/edhoc/`
Documentation: `https://marcovr.gitlab.io/edhoc/`

7.2. Outlook

As the world gets more connected and IoT devices exist for all kinds of use cases, it can only be hoped that security isn't taken lightly. There is a joke that goes "The S in IoT stands for security"[39], which already says a lot about how poorly protected such devices sometimes are. Even though physical attacks can hardly be prevented by software, at least the communication should be properly secured.

CoAP together with OSCORE and EDHOC offers a modern, more power-efficient alternative to Hypertext Transfer Protocol Secure (HTTPS). Although this won't single-handedly solve all security problems with IoT devices, it might still help improve the situation.

A
Common Acronyms

(D)TLS	(Datagram) Transport Layer Security
AAD	additional authenticated data
AEAD	Authenticated Encryption with Associated Data
AES-CCM	AES in CCM mode
AES-NI	AES New Instructions
AES	Advanced Encryption Standard
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BLE	Bluetooth Low Energy
CBC-MAC	Cipher Block Chaining Message Authentication Code
CBOR	Concise Binary Object Representation
CCM	Counter with Cipher Block Chaining Message Authentication Code (CBC-MAC)
CDDL	Concise Data Definition Language
CD	Continuous Delivery/Deployment
CI	Continuous Integration
CoAP	Constrained Application Protocol
COSE	CBOR Object Signing and Encryption
DTLS	Datagram Transport Layer Security
ECDH	Elliptic-curve Diffie–Hellman
EDHOC	Ephemeral Diffie-Hellman Over COSE
ESP-IDF	Espressif IoT Development Framework
GPIO	General-Purpose Input/Output
GPS	Global Positioning System
HKDF	KDF based on HMAC
HMAC-SHA-256	HMAC using SHA with 256 bits hash size
HMAC	Hash-based Message Authentication Code
HTML	Hypertext Markup Language
HTTP(S)	Hypertext Transfer Protocol (Secure)
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDE	Integrated Development Environment
IoT	Internet of Things
ITU	International Telecommunications Union

IV	initialization vector
JOSE	Javascript Object Signing and Encryption
JSON	JavaScript Object Notation
KDF	key derivation function
M2M	Machine-to-Machine
NaCl	Networking and Cryptography library
NFC	Near Field Communication
OSCORE	Object Security for Constrained RESTful Environments
PAD	protected application data
PFS	Perfect Forward Secrecy
PRK	pseudorandom key
REST	Representational State Transfer
RTOS	real-time operating system
SCTP	Stream Control Transmission Protocol
SHA	Secure Hashing Algorithm
SiP	System in Package
SMP	symmetric multiprocessing
SMS	Short Message Service
SRAM	static random-access memory
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UAD	unprotected application data
UDP	User Datagram Protocol
URI	Unified Resource Identifier
USB	Universal Serial Bus
VM	virtual machine
XML	eXtensible Markup Language

References

[1] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The Security Impact of a New Cryptographic Library. In *Progress in Cryptology – LATINCRYPT 2012*, volume 7533, pages 159–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] H. Birkholz, C. Vigano, and C. Bormann. Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures. RFC 8610, RFC Editor, June 2019.

[3] M. Bishop. Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-27.txt, IETF Secretariat, February 2020.

[4] C. Bormann and P. Hoffmann. Concise Binary Object Representation (CBOR). RFC 7049, RFC Editor, October 2013.

[5] Espressif Systems. *ESP32 Datasheet*, October 2016. V1.0.

[6] R. Fielding et al. *Hypertext Transfer Protocol - HTTP/1.1.* 1999. RFC 2616.

[7] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[8] Roy Thomas Fielding. in Information and Computer Science. page 90, 2000.

[9] Urs Gerber. Authentication and Authorization for Constrained Environments. Master's thesis, University of Fribourg (Switzerland), 2018.

[10] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[11] Marc Hadley. Web application description language (wadl). Technical Report TR-2006-153, Sun Microsystems, April 2006.

[12] Ian Jacobs. Uris, addressability, and the use of http get and post. World Wide Web Consortium, TAG Finding, March 2004.

[13] Minhaj Ahmad Khan and Khaled Salah. IoT security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395–411, May 2018.

[14] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges. In *2012 10th International Conference on Frontiers of Information Technology*, pages 257–260, Islamabad, Pakistan, December 2012. IEEE. 2

[15] Andreas Meier. *Relationale und postrelationale Datenbanken.* eXamen.press. Springer, Berlin [u.a.], 6., überarb. und erw. edition, 2007.

[16] Nitin Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, Vienna, Austria, October 2017. IEEE.

[17] Goestenmeyer Ralph and Rehn-Goestenmeier Gudrun. *Das Einsteigerseminar Linux*. Verlag moderne industrie Buch AG & Co. KG, Landsberg, Königswinterer Str. 418, 35227 Bonn, Deutschland, 4., überarb. edition, 2003.

[18] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[19] Andreas Ruppen and Jacques Pasquier-Rocha. A Meta-Model for the Web of Things. Internal Working Paper 13-03, Department of Informatics, University of Fribourg, Switzerland, June 2013.

[20] J. Schaad. Concise Binary Object Representation (CBOR). RFC 8152, RFC Editor, July 2017.

[21] Alexander Schatten. *Best Practice Software-Engineering*. Springer DE, 2010.

[22] G. Selander, J. Mattsson, and F. Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-selander-ace-cose-ecdhe-14.txt, IETF Secretariat, September 2019.

[23] G. Selander, J. Mattsson, F. Palombini, and L. Seitz. Object Security for Constrained RESTful Environments (OSCORE). RFC 8613, RFC Editor, July 2019.

[24] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014.

Referenced Web Resources

[25] Bill Doerrfeld. Is REST Still A Good API Design Style to Use?, October 2019. https://nordicapis.com/is-rest-still-a-good-api-design-style-to-use/ (accessed March 07, 2020).

[26] ESP-IDF Programming Guide. https://docs.espressif.com/projects/esp-idf/en/latest/ (accessed January 21, 2020).

[27] SparkFun ESP32 Thing. https://www.sparkfun.com/products/13907 (accessed February 19, 2020).

[28] ESP-IDF, Fatal Errors. https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/fatal-errors.html (accessed February 27, 2020).

[29] ESP-IDF, FreeRTOS. https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/freertos.html (accessed February 05, 2020).

[30] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (accessed July 30, 2005).

[31] FreeRTOS. https://www.freertos.org/ (accessed February 05, 2020).

[32] Introduction to CI/CD with GitLab. https://docs.gitlab.com/ee/ci/introduction/ (accessed February 06, 2020).

[33] Laurence Goasduff. Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020. https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io (accessed January 15, 2020).

[34] Libcose: Constrained node COSE library. https://github.com/bergzand/libcose (accessed February 10, 2020).

[35] libOSCORE: An OSCORE implementation (not only) for embedded systems. https://gitlab.com/oscore/liboscore (accessed January 21, 2020).

[36] Libsodium documentation. https://libsodium.gitbook.io/doc/ (accessed February 10, 2020).

[37] J. and F. Palombini Mattsson. Comparison of CoAP Security Protocols. https://tools.ietf.org/id/draft-mattsson-lwig-security-protocol-comparison-01.html (accessed January 15, 2020).

[38] Platformio: A new generation ecosystem for embedded development. https://platformio.org/ (accessed February 10, 2020).

[39] Chris Romeo. The S in IoT Stands for Security, June 2017. https://www.iot-inc.com/the-s-in-iot-stands-for-security-article/ (accessed March 20, 2020).

[40] Website of the Sun Microsystems. http://www.sun.com/software/solutions/rfid/ (accessed July 28, 2005).

[41] Nordic Thingy:91. https://www.nordicsemi.com/Software-and-tools/Prototyping-platforms/Nordic-Thingy-91 (accessed February 19, 2020).

[42] TinyCrypt Cryptographic Library. https://github.com/intel/tinycrypt (accessed February 10, 2020).