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

Master Thesis

Martin Andreas Disch

January 2020

Thesis supervisors:

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

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

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

Software Engineering Group

Acknowledgments

My gratitude goes to Prof. Dr. Jacques Pasquier and Arnaud Durand for supervising me. Arnaud has spent a lot of time on interesting discussions, reading and commenting on my draft, as well as giving helpful advice and kind words, for which I am particularly grateful. Thanks are also in order to the authors of EDHOC, most of all John Mattsson, who was a pleasure to work with. Finally, I am indebted to my proofreaders: Nima Yassini, because it is very reassuring to get the opinion of a real professional, and Fabian Allamand, who—despite my recommendation against it—took it upon himself to read this thesis more than once to provide excellent feedback. This report was written with a LaTeX template made by Andreas Ruppen.

Abstract

Securing communication in IoT devices is very important, but also difficult due to constraints in connectivity, processing power, memory size and energy usage. Several lightweight protocols have been developed for this context, among them the EDHOC key exchange and OSCORE, which provides application-layer protection of the commonly used CoAP protocol. These protocols are relatively new and few implementations exist. We implemented an open source library for OSCORE using EDHOC, targeted at embedded devices and written in Rust. This programming language is known for its memory safety, a useful guarantee in security-critical environments. The implementation was demonstrated in a test setup using real hardware, consisting of an embedded resource server, embedded client and a CoAP proxy in between. With this approach we have demonstrated the viability for embedded devices of both the proposed protocols, as well as the Rust programming language, and contributed the first Rust implementations of EDHOC and OSCORE.

Keywords: OSCORE, EDHOC, Rust, Embedded

Table of Contents

1 Introduction

1.1. Motivation

The proliferation of Internet of Things (IoT) devices brings a new set of security challenges with it. On one hand, a never before seen amount of new devices are connected to the Internet, which are prime targets for attackers, due to increasingly becoming part of the fabric of society. On the other hand, these devices are often low-powered and low-performance, making it harder to use existing solutions to secure them. And since security is not yet a selling point in most people's minds, manufacturers have little incentive to spend resources in this area. Maybe most critically, many of these devices will be used for much longer than typical consumer electronics, possibly for decades without ever being updated. Take all of this together, and IoT really could be the asbestos of the future [14].

Many devices are vulnerable by being exposed to the network and using no access control at all, or default credentials. Using old, unencrypted protocols for communication can be a vector of attack as well, leaving the traffic open to examination and manipulation. This is the area we focused on, by implementing a new standard designed to protect a lightweight application protocol commonly used for constrained devices in IoT networks.

1.2. Goal

The goals of this thesis were to:

1. Learn about lightweight protocols and formats used for IoT, including CoAP and CBOR.
2. Get familiar with OSCORE, in particular how CoAP messages are protected and how to establish a shared secret with an authenticated Diffie-Hellman key exchange in EDHOC.

3. Learn about embedded Rust programming, especially about the implications of using Rust for memory management and the limitations when running Rust on bare metal (`#![no_std]`).
4. Write an I/O-free OSCORE library using the EDHOC key exchange for constrained environments using Rust.
5. Write unit tests to ensure compliance of OSCORE/EDHOC messages against test vectors.
6. Write documentation for the API of the library.
7. Demonstrate a protected exchange using a real-world client and resource server running on constrained devices (STM32).

2
Context

2.1. Constrained Networks and Devices

IoT devices are often small, inexpensive and battery-powered, and sometimes use less powerful connectivity than traditional Wi-Fi. As such, they often operate under several constraints, at which point they could fall into the category of *constrained nodes* operating in *constrained networks*. These terms are regularly used in the following chapters, so their definitions as per RFC 7228 [3] are directly quoted here.

Constrained node

"A node where some of the characteristics that are otherwise pretty much taken for granted for Internet nodes at the time of writing are not attainable, often due to cost constraints and/or physical constraints on characteristics such as size, weight, and available power and energy. The tight limits on power, memory, and processing resources lead to hard upper bounds on state, code space, and processing cycles, making optimization of energy and network bandwidth usage a dominating consideration in all design requirements. Also, some layer-2 services such as full connectivity and broadcast/multicast may be lacking."

Constrained network

"A network where some of the characteristics pretty much taken for granted with link layers in common use in the Internet at the time of writing are not attainable."

2.2. IoT Security

IoT security is currently a highly relevant topic and there is no shortage of research on it. In this section, we provide a short overview of the field and give the reader an idea of what kinds of attacks are typical.

A recent survey [10] proposed 9 classes of IoT vulnerabilities:

- *Deficient physical security*: Most IoT devices run in unattended environments and it can be easy for an attacker to get physical access to them, which—as is commonly understood in security research—is game over. It is near impossible to make any guarantees at that point.
- *Insufficient energy harvesting*: IoT devices usually have limited energy, which allows attackers to disable them simply by depleting all of it with a large amount of messages, resulting in Denial-of-Service (DoS).
- *Inadequate authentication*: Complex authentication mechanisms can often not be used on devices with limited energy and computational power. Attackers can exploit insufficient authentication mechanisms, for example to add spoofed malicious nodes.
- *Improper encryption*: Resource limitations could affect the strength of the employed cryptosystems, possibly enabling attackers to break them.
- *Unnecessary open ports*: Many IoT devices leave ports open that run vulnerable services, which are easily exploited by attackers.
- *Insufficient access control*: A common problem is that IoT devices and their cloud management suites do not enforce rules on password complexity or push the user to change the default credentials.
- *Improper patch management capabilities*: The full stack of an IoT system should be updated when necessary, but most manufacturers do not push security patches or do not even have mechanisms in place to do so. Even if update mechanisms are available, they are often insufficiently protected and themselves vulnerable to malicious modifications.
- *Weak programming practices*: Many firmwares have known vulnerabilities such as backdoors or root users by default. Buffer overflow exploits are common.
- *Insufficient audit mechanisms*: There is often a lack of thorough logging procedures, making it impossible to detect malicious activities on a device.

A similar effort [5] also came up with a list of common security issues. Some of them are very relevant to our topic, because they are fully or partly solved by the standards we implemented:

- Authentication and secure communication
- End-to-end security on the transport level
- Session establishment and resumption, where impersonation could be possible
- CoAP security, especially when transported over the Internet

There is no doubt about how valuable IoT devices are to prospective attackers. Being deployed in almost every aspect of modern life, from industrial facilities, to smart cities, agriculture and our homes, they offer a large attack surface and can be used for considerable gain when taken over. But they can also be used to attack other systems. The power of many small devices combined was demonstrated impressively by the Mirai botnet, which was used for unprecedented Distributed Denial-of-Service (DDoS) attacks in 2016 [6].

The ubiquity of devices in the IoT also means that attacks can spread very quickly. Ronen et al. [12] discovered that Philips Hue smartlamps are vulnerable, because their Zigbee (wireless protocol) transmitters had a flaw allowing an attacker to remotely factory reset the devices to take them over. The underlying problem is the insecure design of the

Zigbee Light Link (ZLL) standard. As a result, the devices could be taken over in large numbers, remotely, by driving a car or flying a drone around. The authors were also able to retrieve the global AES key Philips used to encrypt and authenticate new firmware. They proceeded to use both of these to develop a worm, which could spread from device to device after installing a single infected light bulb.

In the case of these smart lights, the manufacturer at least attempted to protect the devices. In some applications that is not even a consideration. For example, many utility companies (electricity, water) install smart meters in homes and use Automatic Meter Reading (AMR) by driving past in a car to efficiently gather usage information. Rouf et al. [13] found that some meters broadcast that information every 30 seconds. The protocol is not secured at all and easily reverse-engineered to read values from homes, or even spoof them. Besides reducing one's utility bill, this could be used to infer the daily routines of residents.

IP cameras are a type of device often exhibiting many different classes of failures. Ling et al. [9] analyzed an Edimax IP camera. Scanning for devices on the network is relatively simple, because the Media Access Control (MAC) address range for a manufacturer is known and the addresses for individual devices can be inferred, since they are often sequential. Active devices can be found by enumerating possible addresses and sending requests, which are answered by active cameras. Once a device is known, bruteforcing the password is an option, because that is not prevented by the camera. Spoofing could also be used to impersonate the device and get the user to enter their credentials on a manipulated login prompt. Even worse, the traffic between the controller and the device is in plain text, so a local attacker could obtain the user password by listening in. Once credentials have been obtained, they can be used to start a Telnet service, which uses username *admin* and password *1234* to provide access to the underlying operating system. Seralathan et al. [18] worked with another unnamed IP camera and found that data was also transferred in plain text. Specifically, the traffic between the controller (mobile phone application) and the cloud service was unencrypted and contained the credentials of all cameras in the system, thereby exposing the passwords used to access them. Cameras are initially set up using the app on the phone, by which the user transmits the Wi-Fi password to the camera, so it can connect itself. This traffic is also not encrypted, allowing a local attacker to obtain the Wi-Fi password and therefore access to the network.

2.3. IoT Connectivity

The requirements for networks connecting IoT devices are very different from those used in most other technology in our daily lives. For example, IoT applications use mostly wireless networks, as the ubiquity of the devices makes wiring infeasible. And since they typically communicate small amounts of information (e.g. sensor readings) at large intervals, the required bandwidth is very small. That is fortunate, as high-bandwidth wireless communication tends to be quite demanding in terms of energy and many of these devices run off batteries, again to make deployment in large numbers and remote locations feasible. It is therefore possible to use specialized wireless communication, which has been designed for low throughput at minimal energy expenditure.

Two important classes of networks for IoT are Low-Rate Wireless Personal Area Networks (LR-WPAN) and Low Power Wide Area Networks (LPWAN).

- LR-WPAN are typically used for shorter range communication of less than 100 m [11]. This type of physical layer is defined by IEEE 802.15.4 and serves as the base for several of the most popular standards defining the upper network layers, such as Zigbee, IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) or Thread, which is itself built on 6LoWPAN [23].
- LPWAN cover a wide geographical area [11]. Examples are Long Range (LoRa), which acts as the physical layer, and Long Range Wide Area Network (LoRaWAN) for the upper network layers, or Narrowband Internet of Things (NB-IoT) [23].

Due to their unique deployments, IoT networks can have a more decentralized topology than a typical home network.

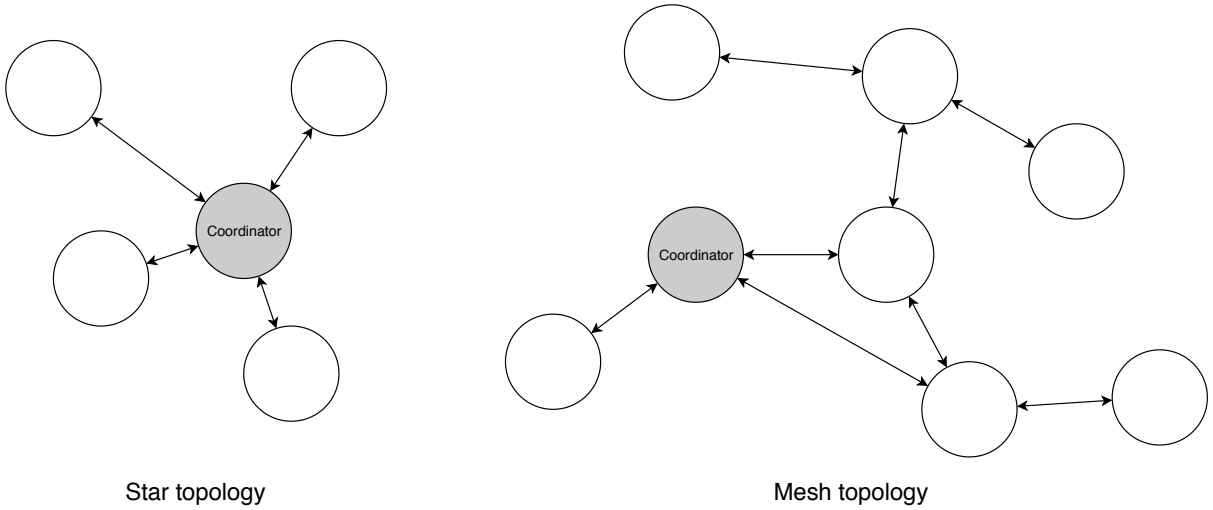

Fig. 2.1.: 6LoWPAN topologies

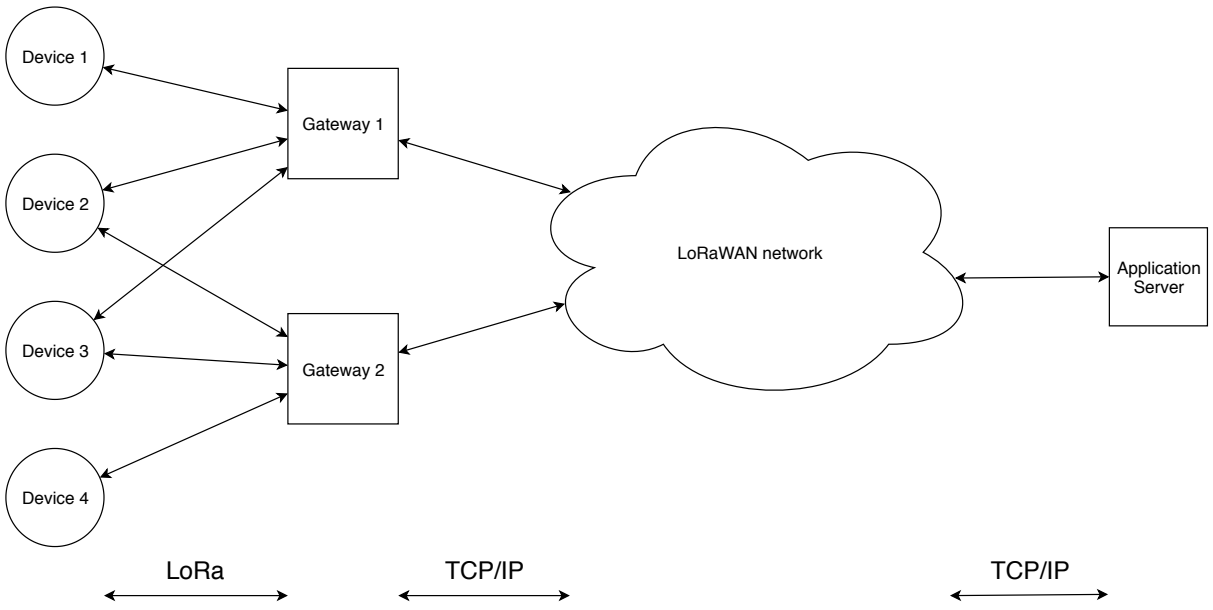

Fig. 2.2.: LoRaWAN topology

It is relatively common in IoT networks that data is routed through intermediary nodes. Since these intermediaries are not necessarily trusted (and trust should be avoided when-

ever possible), there is a strong case for end-to-end protection of sensitive information. While most of these network layers come with some sort of protection, it is neither necessarily end-to-end, nor guaranteed to be there at all in a complex networking stack using different technologies. Thus, there is a need for application layer protocols that provide this level of protection, which is exactly what OSCORE does for CoAP. After this short introduction into IoT, the next chapter will introduce the specific protocols we used and implemented.

3
Background

3.1. CoAP

The Constrained Application Protocol (CoAP) is described in Request for Comments (RFC) 7252 [19]. It is intended as a web transfer protocol for constrained nodes and networks. CoAP can be thought of as an alternative for Hypertext Transfer Protocol (HTTP) for constrained environments. While the Representational State Transfer (REST) architecture (most commonly implemented with HTTP) is a great fit for many use cases in constrained environments, HTTP itself is not. Traditionally, HTTP is very verbose and therefore not efficient to use where limited computation and network resources are available. CoAP aims for much greater efficiency, while still providing a useful request/response model. It achieves this by using a simple binary format and datagram-oriented transports like User Datagram Protocol (UDP), as opposed to more expensive connection-oriented transports such as Transmission Control Protocol (TCP). It can still provide

optional reliability through the use of message IDs. Finally, it is transport-agnostic and HTTP-mappable, meaning proxies can easily provide access to CoAP resources via HTTP.

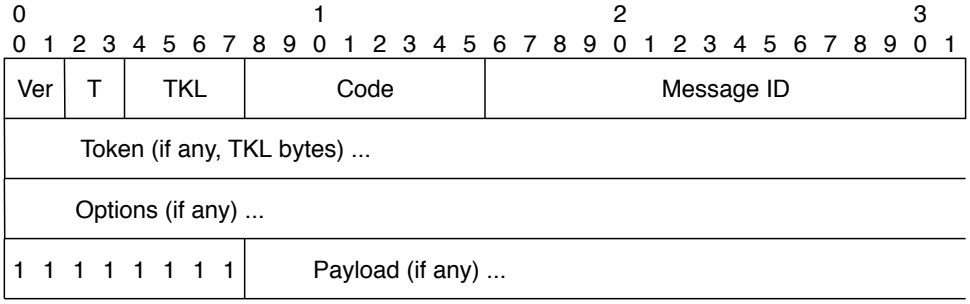

Fig. 3.1.: CoAP Message Format. Adapted from [19].

The message format, as shown in Figure 3.1, is fairly straightforward. It starts with a 4-byte header, containing the CoAP version, message type (e.g. `Confirmable`, `Acknowledgement`), token length, the code (e.g. `GET`, `Changed`) and the message ID. After that follows a variable-length token (0 to 8 bytes), used for mapping requests to responses. This is followed by any number of options with associated values, like `Uri-Path` or `Content-Format`. If there is an optional payload, it makes up the rest of the message and is separated from the options by a one-byte payload marker (`0xFF`).

Like HTTP, CoAP is not encrypted by default, but a binding to Datagram Transport Layer Security (DTLS) is specified.

3.2. CBOR

The Concise Binary Object Representation (CBOR) is described in RFC 7049 [4]. It relates to JavaScript Object Notation (JSON) in a similar way as CoAP does to HTTP, in that it is mostly a more efficient format with the same goal. It is also a binary format and mappable to JSON in this case. Because the more compact representation results in smaller message sizes, it is sometimes used instead of JSON in constrained environments.

The encoding specifies that the initial byte of each data item contains information about the *major type* (3 most significant bits) and additional information (5 least significant bits). Examples of major types are unsigned integers, negative integers, byte strings, text strings, arrays or maps. The additional information can indicate the length of the value (how many of the following bytes belong to it), or—for small integers—be the value itself.

A diagnostic notation for CBOR is also defined, which is loosely based on JSON. As a practical example for CBOR data items, consider the following structure used in EDHOC.

```
M_V =
[
  "Signature1",
  << { 4: h'a3' } >>,
  h'5550b3dc5984b0209ae74ea26a18918957508e30332b11da681dc2afdd870355',
  << {
    1:  1,
   -1:  6,
   -2:  h'1b661ee5d5ef1672a2d877cd5bc20f4630dc78a114de659c7e504d0f529a6b
```

```
10            d3'
11   } >>
12 ]
```

List. 3.1: EDHOC Sig_structure

It is an array with four elements, starting with a text string. The second element is a map using an integer as key and a byte string as value. However, this map is not present as the direct encoding of a CBOR map. Instead, it is additionally wrapped in a byte string, indicated by the `<< ... >>` notation. This means that the map is encoded to its normal CBOR representation (the bytes `0xA10441A3`), but prepended by a byte saying that it is a byte string of length four, meaning that the value of this array entry is then `0x44A10441A3`. This wrapping may seem superfluous, but is the result of how some structures are defined in COSE, which is used in EDHOC. The other elements in the array use the same types.

```
1 message_1 =
2 (
3   1,
4   0,
5   h'b1a3e89460e88d3a8d54211dc95f0b903ff205eb71912d6db8f4af980d2db83a',
6   h'c3'
7 )
```

List. 3.2: EDHOC message

Listing 3.2 demonstrates CBOR sequences [2], an extension to CBOR, which is still a work in progress. Since CBOR data items are self-delimiting, it is possible to simply concatenate any number of them, while still being able to consume them one by one. That is exactly what a CBOR sequence is, indicated by parentheses `( ... )`. By using sequences over arrays, no additional byte indicating an array of a certain length is necessary, thus reducing the message length.

3.3. COSE

CBOR Object Signing and Encryption (COSE) is described in RFC 8152 [15]. Like all standards so far, it too is a more lightweight alternative to an existing solution, in this case Javascript Object Signing and Encryption (JOSE). Among other things, COSE specifies how to process encryption and signatures and encode keys using CBOR. It is relevant here, because it is used by EDHOC for signatures, encryption and key encoding. But since the EDHOC specification contains all COSE structures required, it is not necessary to be overly familiar with it. Besides pointing out that the CBOR data item from Listing 3.1 is in fact a COSE `Sig_structure`, we will not delve deeper into it.

3.4. EDHOC

3.4.1. Overview

Ephemeral Diffie-Hellman Over COSE (EDHOC) is described in [16]. In the words of the authors, it is "a very compact, and lightweight authenticated Diffie-Hellman key exchange

with ephemeral keys". It extends the theoretical SIGMA-I protocol [7] with elements like transcript hashes and cipher suite negotiation, to turn it into a "full-fledged" protocol. It reuses COSE for cryptography, CBOR for encoding and CoAP for transport, although that is merely a recommendation, as EDHOC can use any transport. These are reused in the hope of reducing the additional code footprint of an implementation. Additionally, EDHOC is designed to have small message sizes, which can be sent over a small number of frames in the network, conserving energy. These properties make it an ideal choice for constrained environments.

Two cipher suites are currently defined. To keep code size small, constrained devices often only support one. We therefore implemented the mandatory cipher suite, which uses:

- AES-CCM-64-64-128 as the Authenticated Encryption with Associated Data (AEAD) algorithm
- ECDH-SS + HKDF-256 as the Elliptic Curve Diffie-Hellman (ECDH) and HMAC-based Extract-and-Expand Key Derivation Function (HKDF) algorithms
- Curve25519 as the ECDH curve
- Ed25519 (which is an Edwards-Curve Digital Signature Algorithm (EdDSA) using SHA-512 on Curve25519) as the signature algorithm

EDHOC can be authenticated asymmetrically by Raw Public Keys (RPK) or public key certificates, and symmetrically by Pre-Shared Keys (PSK). We implemented the RPK case, so that is what we will look at. Here, the only requirement for using EDHOC is that each party owns a private key, of which the other party can retrieve the accompanying public key, given its identifier. These need to be established out-of-band. Once an EDHOC exchange is complete, both parties are in possession of the same *master secret* and *master salt*, from which they can derive shared application keys. The main use case for EDHOC is establishing an OSCORE *security context*.

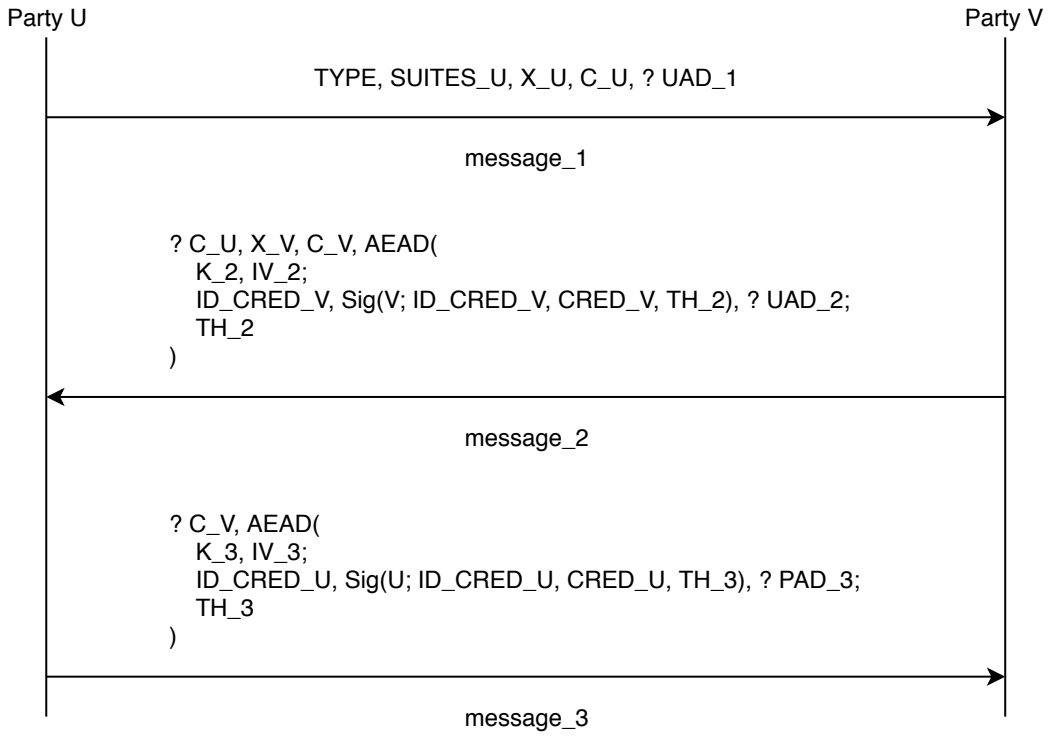

Fig. 3.2.: EDHOC overview with asymmetric key authentication. Adapted from [16].

Figure 3.2 shows the three messages that make up a successful EDHOC exchange. Optional values are labeled with question marks.

3.4.2. message_1

Party U sends a message consisting of the following:

- `TYPE` is an integer giving information about the correlation properties of the underlying transport. Some transports—such as CoAP—have features like the CoAP token, which allow matching two messages (typically request and response) with each other. This is important when doing simultaneous EDHOC exchanges with multiple peers, to be able to know which received message belongs to which exchange. If the transport allows for some correlation, the explicit connection identifiers within the EDHOC messages may be omitted, reducing the message size.
- `SUITES_U` is an array of supported cipher suites, or just a single value in case there is only one, to further reduce the message size.
- `X_U` is the x-coordinate of the ephemeral public key that Party U generated for this exchange.
- `C_U` is the connection identifier chosen by Party U for this exchange.
- `UAD_1` is optional, unprotected opaque application data.

After receiving it, Party V verifies that it can work with the chosen parameters, responding with an error message if that is not the case. Otherwise, it constructs `message_2`.

3.4.3. message_2

- C_U is Party U's connection identifier, included only if the transport does not allow Party U to correlate the message with the first one.
- X_V is the x-coordinate of the ephemeral public key that Party V generated for this exchange.
- C_V is the connection identifier chosen by Party U for this exchange.
- K_2 is a symmetric encryption key derived from the shared secret established with the ephemeral Diffie-Hellman exchange (which is now complete for Party V, since it has received Party U's public key in the first message), as well as TH_2 (explained below).
- IV_2 is a nonce also derived from the ECDH shared secret and TH_2.
- ID_CRED_V is the identifier by which Party U can retrieve the public authentication key of Party V.
- V is the private authentication key of Party V.
- CRED_V is a credential containing the public authentication key of Party V.
- TH_2 is a transcript hash made from message_1 and the data Party V sends in this message (C_U if present, X_V, C_V).
- UAD_2 is optional, unprotected opaque application data.
- Sig(...) is a signature made with Party V's private authentication key of ID_CRED_V, TH_2 and CRED_V.
- AEAD(...) is AEAD using K_2 and IV_2, with the plaintext consisting of ID_CRED_V, the previous signature and optionally UAD_2, and the associated data being TH_2.

Party U decrypts and verifies the AEAD ciphertext in message_2. It can do so, because it possesses all the information used by Party V to create it (ECDH shared secret, TH_2, etc.). It also verifies the enclosed signature with the public authentication key of Party V. If any verification step fails, it responds with an error message. Otherwise, it constructs message_3, which is very similar to message_2.

3.4.4. message_3

- C_V is Party V's connection identifier, included only if the transport does not allow Party V to correlate the message with the first one.
- K_3 is a symmetric encryption key derived from the shared secret established with the ephemeral Diffie-Hellman exchange, as well as TH_3 (explained below).
- IV_3 is a nonce also derived from the ECDH shared secret and TH_3.
- ID_CRED_U is the identifier by which Party V can retrieve the public authentication key of Party U.
- U is the private authentication key of Party U.
- CRED_U is a credential containing the public authentication key of Party U.
- TH_3 is a transcript hash made from TH_2, the ciphertext from message_2 and the data Party V sends in this message (C_V if present).

- `PAD_3` is optional, protected opaque application data. It is protected, because at this point Party U already has all the information needed to derive symmetric application keys.
- `Sig( ... )` is a signature made with Party U's private authentication key of `ID_CRED_U`, `TH_3` and `CRED_U`.
- `AEAD( ... )` is AEAD using `K_3` and `IV_3`, with the plaintext consisting of `ID_CRED_U`, the previous signature and optionally `PAD_3`, and the associated data being `TH_3`.

Party V decrypts and verifies the AEAD ciphertext in `message_3`. It also verifies the enclosed signature with the public authentication key of Party U. If any verification step fails, it responds with an error message.

3.4.5. Completion

The transport may cause a final acknowledgment message to be sent, as is the case when using CoAP. Because EDHOC is sent in `Confirmable` CoAP messages, requests are always answered by an ACK, which can contain a "piggybacked response". This is the case for `message_2`, as shown in Figure 3.3.

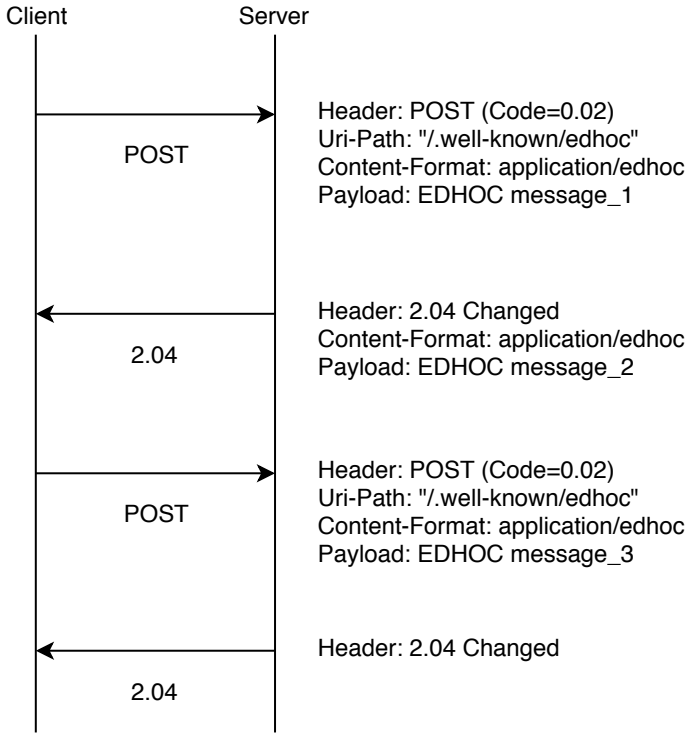

Fig. 3.3.: Transferring EDHOC in CoAP. Adapted from [16].

EDHOC does not define a message after `message_3`, but CoAP will still send an empty ACK, after which the exchange can be considered finished. Both parties calculate `TH_4` from `TH_3` and the ciphertext from `message_3`. This is then used in the *EDHOC-Exporter* interface, which takes `TH_4` and the ECDH shared secret and can be used to derive application keys and other data of any length. For OSCORE, the master secret and master salt are generated in this manner.

3.5. OSCORE

3.5.1. Overview

Object Security for Constrained RESTful Environments (OSCORE) is described in RFC 8613 [17]. It does application-layer protection of CoAP, providing end-to-end protection, but still allowing for proxy operations. This is an important distinction, as DTLS only does hop-by-hop protection, exposing the information to intermediary nodes, subject to examination and manipulation. It works directly on CoAP (or HTTP, since CoAP can be transported in HTTP) messages. This means that the resulting OSCORE messages are still valid CoAP (or HTTP, but we will focus on CoAP from now on) messages with the same characteristics, for example with respect to the underlying transports. As such, OSCORE retains the suitability of CoAP for constrained environments, keeping the message size and additional code requirements small.

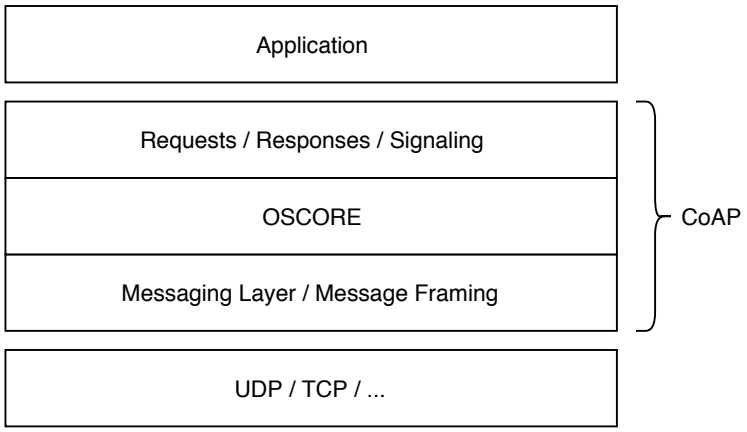

Fig. 3.4.: Abstract layering of CoAP with OSCORE. Adapted from [17].

As shown in Figure 3.4, OSCORE protects the RESTful interactions—such as the request method, requested resource and payload—but not the CoAP messaging layer, because that can change between endpoints and needs to be accessible for proxy operations.

3.5.2. Security Context

First and foremost, OSCORE needs to establish a secure channel between two parties using symmetric encryption. To do so, it requires shared keys. These can be derived from a master secret and master salt established out-of-band with EDHOC, just a master secret, or additionally with something called an *ID context*. Since we already have EDHOC, we implemented the case of using both a master secret and master salt which it provides, and will describe this approach.

To keep track of the cryptographic operations, OSCORE introduces the *security context*, which consists of three parts:

- The *common context* is the same for both. For simplicity, we can consider that it contains only one relevant piece of information, the *common IV*. It is derived from the master secret and master salt and used to compute the AEAD nonce.
- The *sender context* contains

  - The *sender ID*, which is used to identify the sender context that generated an encrypted message, derive AEAD keys, as well as act as an input to the AEAD nonce.
  - The *sender key* is the symmetric key used to send messages and is derived from the master secret, master salt and sender ID.
  - The *sender sequence number* is incremented with every sent message and used as the partial IV to generate unique AEAD nonces.
- The *recipient context* contains
  - The *recipient ID*, which is used to identify the recipient context with which to decrypt a received message, derive AEAD keys, as well as act as an input to the AEAD nonce.
  - The *recipient key* is the symmetric key used to receive messages and is derived from the master secret, master salt and recipient ID.
  - The *replay window* is only present on a server and is used to make sure no recently received messages are replayed.

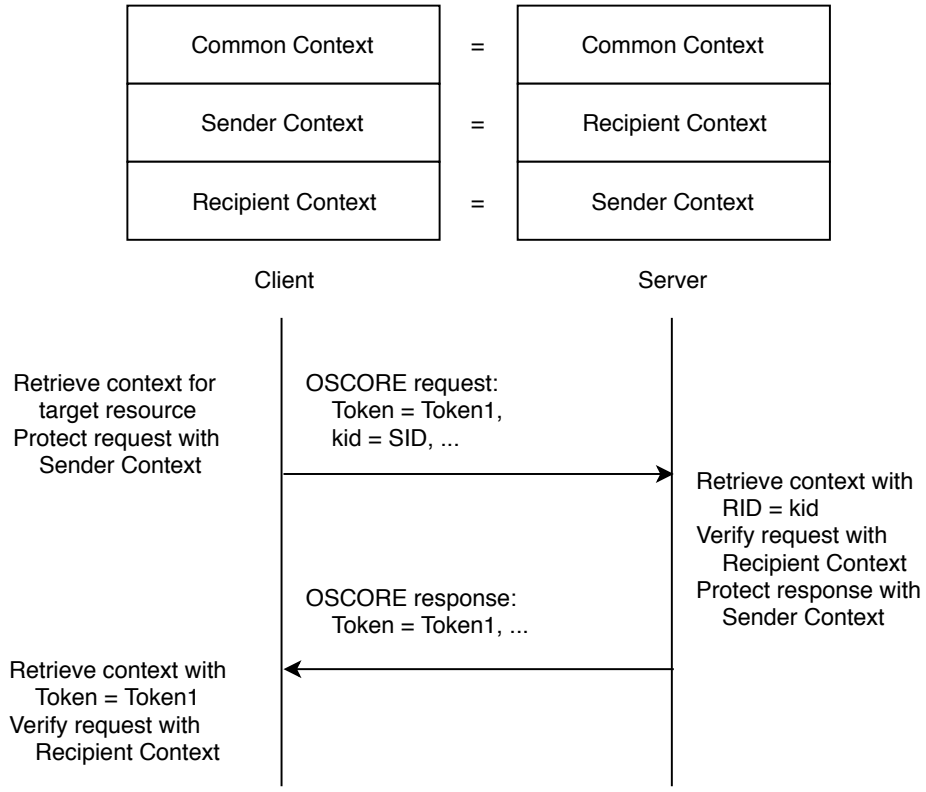

Fig. 3.5.: Retrieval and use of the security context. Adapted from [17].

As shown in Figure 3.5, the sender context of one party is almost the same as the recipient context of the other one, since this is symmetric encryption. After a successful EDHOC exchange, both parties are able to construct these mirrored security contexts, since they possess all required information: the master secret, master salt and both their own ID (sender ID) as well as their peer's (recipient ID), which could be the connection identifiers used in the EDHOC exchange or the key IDs from the public authentication keys it uses.

With that, they could now protect arbitrary request/response communication between them. Since it is very important to use a unique nonce for every message sent with a key in symmetric encryption, a new AEAD nonce is generated for every request. It is derived from the partial IV (the sender sequence number), the sender ID and the common IV. The partial IV is the only part of this that the other endpoint does not already possess. Therefore, it needs to be sent together with the message, as does the sender ID, so the appropriate recipient context can be retrieved to decrypt the message. The associated data for the AEAD are the OSCORE version and algorithms used, as well as the sender ID and the partial IV. The former two are present to make sure both sender and receiver use the same version and algorithms, the latter to protect them against manipulation, since they are not encrypted.

The receiver can then retrieve the recipient context, reconstruct the associated data from the recipient ID (= sender ID of the other party) and partial IV it sent along, compute the AEAD nonce from the partial IV, the recipient ID and the common IV and use these to both decrypt and verify the encrypted data. Protecting the response is almost the same as for a request, except for two differences:

- It is possible to reuse the AEAD nonce from the request (since it is used with a different key, this is not a problem), in which case no partial IV is sent in the response. If a new nonce is generated, the sender sequence number is incremented and sent as the partial IV, as is done when sending a new request.
- It is not necessary to send the sender ID, because the receiver can retrieve the appropriate recipient context from the CoAP token of the enclosing CoAP message, which is the same one it has originally chosen for the request.

Verification of the response is also very similar to that of a request, except for accommodating the two previously mentioned differences in a response. This is roughly how the core cryptographic operations in OSCORE work, but this is only one half. Next, we will look at how this is applied to protect CoAP messages.

3.5.3. CoAP Protection

As we mentioned previously, OSCORE protects as much of the CoAP message as possible, while still allowing proxy operations. In principle, this means protecting three important parts of a message:

- The code, because we would not want a malicious proxy to be able to manipulate a `GET` into a `DELETE`, for example.
- Some sensitive options, because again, a proxy manipulating the query or knowing which resource is requested could be dangerous.
- The payload, which we obviously want to keep confidential as well.

In order to do so, CoAP options are split into two categories, as shown in Table 3.1.

No.	Name	E	U
1	If-Match	x	
3	Uri-Host		x
4	ETag	x	
5	If-None-Match	x	
6	Observe	x	x
7	Uri-Port		x
8	Location-Path	x	
9	OSCORE		x
11	Uri-Path	x	
12	Content-Format	x	
14	Max-Age	x	x
15	Uri-Query	x	
17	Accept	x	
20	Location-Query	x	
23	Block2	x	x
27	Block1	x	x
28	Size2	x	x
35	Proxy-Uri		x
39	Proxy-Scheme		x
60	Size1	x	x
258	No-Response	x	x

Tab. 3.1.: Protection of CoAP options [17]

Class U options remain untouched in the CoAP message, while Class E options are moved into the payload, which is protected by the AEAD. A good example of the two classes are the `Uri-Host` and `Uri-Path` options. `Uri-Host` cannot be protected, since the destination of a CoAP message must be known. The `Uri-Path` on the other hand—specifying which exact resource is requested—can be encrypted and is only decrypted at the final destination, the resource server.

There are a few CoAP options that receive special treatment in OSCORE, among them the ones related to proxying. If a normal CoAP message were to be sent over a proxy `proxy.com` with the final destination of `coap://example.com/resource?q=1`, then it would have the `Uri-Host` option set to `proxy.com`, since that is the first destination. It would contain the full Uniform Resource Identifier (URI) of `coap://example.com/resource?q=1` in the `Proxy-Uri` option, to be used by the proxy to update the message before passing it along. With OSCORE, the `Proxy-Uri` option is taken apart and both the `Uri-Path` and `Uri-Query` part of it are removed and put into their respective options, as the proxy would do in plain CoAP. The difference is that in OSCORE, they are Class E options and therefore protected and invisible to the proxy. The proxy can only see `coap://example.com` in the `Proxy-Uri` option, which it uses to deliver the message. At the receiving end, the OSCORE message is turned into a regular CoAP message again by decrypting the protected `Uri-Path` and `Uri-Query` options, restoring them and thus turning it into the same CoAP message the server would have received if no protection had been applied.

Besides certain CoAP options, the payload and code (equivalent of HTTP method) are also protected.

Field	E	U
Version (UDP)		x
Type (UDP)		x
Length (TCP)		x
Token Length		x
Code	x	
Message ID (UDP)		x
Token		x
Payload	x	

Tab. 3.2.: Protection of CoAP header fields [17]

Both are encoded into the new, encrypted payload as well, together with the Class E options. The CoAP code is replaced by a dummy code—`POST` for requests and `Changed` for responses.

Now we can look at how all of this is encoded into a CoAP message.

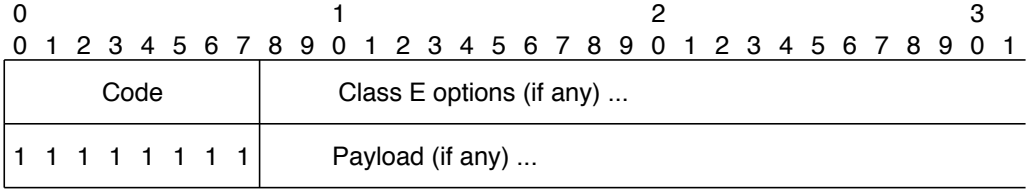

Fig. 3.6.: Plaintext. Adapted from [17].

The code, Class E options and payload are formatted similarly to a real CoAP message, except with a reduced header (code only), as shown in Figure 3.6. This is the plaintext which—after being encrypted to ciphertext and authenticated with associated data as described in Section 3.5.2 with the AEAD—will be the payload of the updated CoAP/OSCORE message.

From Section 3.5.2 we recall that the cryptographic operation of OSCORE requires the partial IV and sender ID (the key ID of the constructing sender context) to be sent along. For this, a new CoAP option is introduced, the `OSCORE` option.

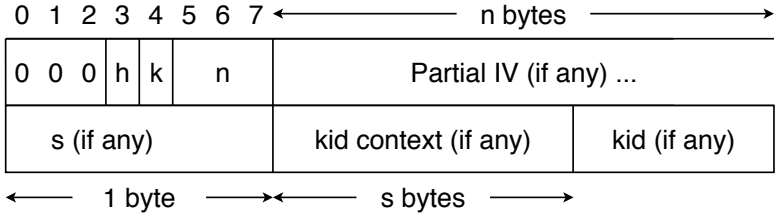

Fig. 3.7.: The `OSCORE` option value. Adapted from [17].

Figure 3.7 shows how they are encoded in the value of that option. For further information, refer to Section 6.1 of RFC 8613 [17].

With that, OSCORE has all the necessary features to provide end-to-end protection of CoAP messages between a client and a server, by transforming the original message to protect sensitive parts, such as the code, a subset of the options and the payload.

3.6. Language Choice

Rust is a new multi-paradigm systems programming language, which had its first stable release in 2015 [21]. Arguably the biggest benefit of Rust is its ability to combine the traditionally conflicting goals of performance and safety, particularly memory safety. While some languages use garbage collection to achieve this, it comes at the cost of requiring a rather extensive runtime and has a negative effect on performance. Rust, on the other hand, has a minimal runtime like those of C and C++, which are so small that the languages are often said to have "no runtime".

Rust achieves its memory safety by using the concept of *ownership*. In a nutshell, this means that all data is always owned by some variable. This data can be *moved* to a different owner like a structure or a variable inside a function, which at this point becomes the new owner, making the data unavailable to the previous one.

We can illustrate this with a simple example.

```
1  struct Point {
2      x: u32,
3      y: u32,
4  }
5
6  fn plot(p: Point) {
7      // Do something
8  }
9
10 fn main() {
11     let x = 25;
12     let y = 10;
13
14     let p1 = Point { x, y };
15
16     plot(p1);
17 }
```

List. 3.3: Example 1

We have a structure `Point` and a function that takes a `Point` as an argument. In our application logic, we define two integer values, use them to construct a `Point` and pass it to the function. There is nothing unusual about this.

But if we introduce another variable and assign the value of our point to it, we run into trouble.

```
1  fn main() {
2      let x = 25;
3      let y = 10;
4
5      let p1 = Point { x, y };
6      let p2 = p1;
7
8      plot(p1);
9  }
```

List. 3.4: Example 2

The compiler will prevent us from doing that.

```
error[E0382]: use of moved value: `p1`
  --> src/main.rs:20:10
   |
14 |     let p1 = Point {
   |         -- move occurs because `p1` has type `Point`, which does not implement
   |            the `Copy` trait
...
18 |     let p2 = p1;
   |              -- value moved here
19 |
20 |     plot(p1);
   |          ^^ value used here after move
```

List. 3.5: Move error message

Listing 3.5 is the perfect example for the famously helpful error messages of the Rust compiler. It tells us exactly what the problem is: `let p2 = p1` transfers ownership of the point to the `p2` variable and `p1` is invalidated. The point is moved, which is exactly what the compiler tells us.

To illustrate what it means by "does not implement the `Copy` trait", here is another example.

```rust
fn main() {
    let x = 25;
    let y = 10;
    let z = x;

    let p1 = Point { x, y };
}
```

List. 3.6: Example 3

Based on what we just learned, this should not compile either. It is the exact same case, except that this time `z` is assigned the value of `x`, but `x` is used nonetheless in constructing a point, which should not work, because its value has been moved. But it does. The reason is that integers are a `Copy` type, meaning they implement the `Copy` marker trait to indicate that they can be trivially copied, without any side effects. So effectively, the value is simply copied and both variables remain valid and independent. `Point`, on the other hand, is an *owned* type, so it is moved. As an aside, because `Point` consists entirely of `Copy` types (the integers), it could trivially be made `Copy` as well, which would let us compile the example from Listing 3.4. But for the sake of our demonstration, we want it to be an owned type.

Ownership works for function arguments too.

```rust
fn plot(p: Point) {
    // Do something
}

fn main() {
    let x = 25;
    let y = 10;

    let p1 = Point { x, y };

    plot(p1);
```

```
    plot(p1);
}
```

List. 3.7: Example 4

This will give us the same kind of error as before.

```
error[E0382]: use of moved value: `p1`
  --> src/main.rs:17:10
   |
14 |     let p1 = Point { x, y };
   |         -- move occurs because `p1` has type `Point`, which does not implement
   |            the `Copy` trait
15 |
16 |     plot(p1);
   |          -- value moved here
17 |     plot(p1);
   |          ^^ value used here after move
```

List. 3.8: Move error message

That is because the `plot` function takes ownership of its argument, so it cannot be used afterwards, as `p1` is moved into `plot`. Because there is no reason for `plot` to take ownership of the argument, the solution here is to make it accept a *reference* instead.

```
fn plot(p: &Point) {
    // Do something
}

fn main() {
    let x = 25;
    let y = 10;

    let p1 = Point { x, y };

    plot(&p1);
    plot(&p1);
}
```

List. 3.9: Example 5

In this implementation, `plot` takes a reference to a `Point`, thereby *borrowing* it. The *borrow checker* can verify that this happens in a safe way, namely that at no time there is more than one mutable reference to the data.

To demonstrate this, we change the example slightly.

```
fn main() {
    let x = 25;
    let y = 10;

    let mut p1 = Point { x, y };

    let ref1 = &p1;
    let ref2 = &p1;

    plot(ref1);
    plot(ref2);
```

```
}
```

List. 3.10: Example 6

Semantically, this is almost the same as before. In preparation for the next example, we declare `p1` as *mutable* (by default, variables are *immutable* in Rust), which means we could modify it, if we wanted. We do not do that, instead we simply create two immutable references to it, which is safe and therefore allowed. But if we try to create a mutable reference in addition, the compiler once again complains.

```
fn main() {
    let x = 25;
    let y = 10;

    let mut p1 = Point { x, y };

    let ref1 = &p1;
    let ref2 = &p1;
    let ref3 = &mut p1;

    plot(ref1);
    plot(ref2);
}
```

List. 3.11: Example 7

```
error[E0502]: cannot borrow `p1` as mutable because it is also borrowed as immutable
  --> src/main.rs:18:16
   |
16 |     let ref1 = &p1;
   |                --- immutable borrow occurs here
17 |     let ref2 = &p1;
18 |     let ref3 = &mut p1;
   |                ^^^^^^^ mutable borrow occurs here
19 |
20 |     plot(ref1);
   |          ---- immutable borrow later used here
```

List. 3.12: Mutable borrow error

We cannot safely have mutable and immutable references at the same time, so the compiler prevents us from shooting ourselves in the foot.

The compiler does a great many things more, explaining all of which is out of scope for this thesis. Interested readers are better served looking into the excellent Rust book[1] to learn more. But to mention a few more highlights, the compiler also does not allow having uninitialized data, like trying to use an integer that was only declared and not defined, or having an uninitialized array. As with borrowing, all this verification happens at compile time, so there is no impact on performance. Finally, when the owner of some data goes out of scope, it is dropped automatically. Thanks to this concept of Resource Acquisition Is Initialization (RAII), the programmer does not have to free memory, even though there is no garbage collection involved. Furthermore, it may have occurred to the astute reader

[1] https://doc.rust-lang.org/book/

that ownership also lends itself well to preventing data races in multi-threaded environments. Indeed, Rust makes it exceptionally easy to write correct concurrent applications, which is usually a hard problem.

To wrap up, here are some of the other important features of Rust:

- It is extensively built around zero-cost abstractions that make it easier to use, but are optimized out at compile time and do not affect performance.
- It does generics with traits and favors composition over inheritance.
- It has powerful pattern matching.
- It has good interoperability with C, making it fairly easy to both create libraries to be used from C code and use existing C libraries in Rust.

Having pointed out many of Rust's advantages, we should also mention the downsides. While there are of course some, it seems to have come closer to being ideal than almost all programming languages before it. Many seem to agree, as Rust has been chosen the most loved programming language in the 2019 edition of the Stack Overflow Developer Survey for the fourth year in a row [20]. Probably the most obvious tradeoff of the language is related to its biggest strength—that it can be completely memory safe without sacrificing performance. While that is true, nothing is ever really free. In the case of Rust, the cost is the steeper learning curve that comes with ownership. Beginners often struggle against the compiler complaining (in admittedly very helpful error messages) about things that are possible in other languages, but forbidden in Rust, because they are not verifiably correct and rely on the programmer to "get it right". But the fact that Rust is a bit harder to learn is offset by great resources, a friendly community and the many useful features it offers in return. It is also worth pointing out that Rust is in many ways still a simpler language than C++, which has become quite large over the years.

With this basic knowledge of the properties of the language, we are now equipped to ask the question of where it makes most sense to use Rust. With memory safety being the most prominent feature of the language, applications where getting this wrong and having dangling pointers or missing bounds checks can lead to severe security issues are the most obvious place to start. After all, about 70% of security vulnerabilities across all Microsoft products that are assigned a Common Vulnerabilities and Exposures (CVE) number, are memory safety issues, which would not have occurred in Rust code [8]. Examples where Rust has improved security are a PGP key server[2] or a parser for MP4 metadata used in Firefox[3]. Another component of Firefox that has been replaced with Rust code is the CSS engine[4]. In this case, the main goal was not security, but performance improvements through parallelization, which was made much easier by Rust's concurrency model. In theory, being a systems programming language, Rust can be used for virtually anything. It is as powerful as C and can be used to write kernels and full operating systems, but from a design point of view, it is more closely related to C++ with its higher level abstractions. Thanks to that design and its crate (library) ecosystem, it is even possible to write code that is almost as simple as Python at times.

Having established that we can write fast and safe code with Rust, we really do not need any more reasons to also want to use Rust for programming embedded devices,

[2] `https://keys.openpgp.org/`

[3] `https://github.com/mozilla/mp4parse-rust`

[4] `https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/`

particularly when they do something related to security, such as OSCORE. In fact, Rust's static guarantees are especially useful when working with hardware. It allows the compiler to verify that there is only one reference to a particular piece of hardware at all times, that hardware can only be used once it's been properly initialized, or that operations can only be performed on correctly configured peripherals. To support embedded devices and other specialized contexts, Rust provides a subset of its functionality specifically for bare metal environments. Usually, a program is run in an environment with an operating system, which allows for interaction with file systems, threads, networking and similar functionality. Like most programming languages, Rust's standard library provides abstractions for these. On bare metal though, there are no such things. That is why Rust has `libcore`, a platform-agnostic subset of the standard library. This is where `#![no_std]` comes in. It is an attribute used to indicate to the compiler that a program or library should not load the standard library. It is also used as a term to convey that some code does not require the standard library and is therefore suitable for embedded development. For example, we frequently mention "`#![no_std]` libraries", such as the one we built. Due to this capability and the efforts of the community—especially Jorge Aparicio[5]—Rust is starting to become a viable option for programming embedded devices. There is already great tooling and a vibrant ecosystem of device crates that support a specific device or device family. Hardware Abstraction Layer (HAL) implementations which implement traits from the `embedded-hal`[6] crate can then be built on these device crates. This allows for generic drivers to be written, that can work with any device for which there is such a HAL implementation. Finally, there are also board support crates, which give abstracted access to functionality of a specific board. The Cortex-M family of processor cores is already well supported, as is the STM32 family of integrated circuits using them and the STM32 Discovery boards that we used in our experimental setup.

[5] `https://github.com/japaric`
[6] `https://docs.rs/embedded-hal`

4
Implementation

4.1. EDHOC

4.1.1. Approach

The implementation work started with EDHOC, because it is relatively self-contained and can therefore be implemented separately, without knowing too much about the larger ecosystem of related RFCs. The general approach was to go step by step along the EDHOC exchange, starting with the encoding of the first message, then the decoding and verification thereof, moving on to generating the second message and so on. This first part of the implementation produced many utility functions, each concerned with one task. For instance, some of them are responsible for computing various transcript hashes or generating one of the required COSE structures. Other functions use these to provide their own functionality, such as the one building the second message. When the library was implemented, the EDHOC draft was at version 13 and did not yet contain any test vectors, which made the process a bit more difficult. Instead, we used the given examples and created our own test vectors to be able to unit test virtually every function that was created. This increased the odds of success when they would all be used together for the first time in a full test run. Each time a step like verifying the second message was completed, the as of yet incomplete test run was extended with this functionality, so that when the last one was finished, we already had a test script for a full EDHOC run.

4.1.2. AES-CCM

One problem we ran into was the lack of an AES-CCM implementation. There is a pure Rust Advanced Encryption Standard (AES) crate[1], but no implementation of Counter with CBC-MAC (CCM) mode, which makes it an AEAD and is part of the mandatory cipher suite in EDHOC and the default algorithm in OSCORE. Because we could not implement either of them in a compliant way without CCM mode, we had to build our own.

We started from Intel's TinyCrypt library[2]. It provides some cryptographic primitives, optimized for small code size, not performance. This results in very neat and concise code—for example, the CCM implementation is only 160 lines of C code. We ran that through a Rust transpiler, which gave us ugly, but (after a few modifications) running code that we could successfully verify against the test vectors. That allowed us to replace it function by function with a proper, handwritten port of TinyCrypt's CCM implementation and at each step run it against the test vectors to make sure no errors were introduced. In the end, we had a fully compliant AES-CCM implementation for bare metal targets[3].

There appears to be some interest from the community in using this library. Thanks to the contribution of a third party, it now implements the `Aead` trait[4], which is a trait exposed by the de facto collection of cryptographic algorithms written in Rust. Its intention is to give a common interface to as many AEAD implementations as possible, enabling their use as drop-in replacements for each other, greatly increasing interoperability.

4.1.3. Draft Contributions

While working on EDHOC, we spotted several issues in the draft, from inconsistent naming and typos to wrong CBOR encoding in some examples. There were also a few questions about certain aspects, so we decided to create a pull request with our editorial fixes on the working group's draft repository, and use the opportunity for clarification. Our fixes were appreciated, questions answered, and since work was ongoing to create a script to generate test vectors, we were given a preliminary version to verify. This was very helpful, since we were able to discover mistakes in our implementation as well as the test vectors themselves. At the time of writing, the draft is at version 14—with test vectors—our code is still compliant, and the test vectors have been independently verified against our implementation. Being able to contribute in a small way to the evolution of the draft was a nice and unexpected experience and benefited our implementation efforts greatly.

4.1.4. API Design

Once all the implemented primitives were shown to behave correctly and combined into a complete EDHOC run, we wrapped them up in a straightforward Application Programming Interface (API). Since EDHOC is split distinctly into two roles—Party U and Party

[1] `https://crates.io/crates/aes`
[2] `https://github.com/intel/tinycrypt`
[3] `https://crates.io/crates/aes-ccm`
[4] `https://docs.rs/aead/0.2.0/aead/trait.Aead.html`

V—with different operations, it makes sense to have two types, one for each. Because this is effectively a protocol implementation, operations happen in a strict order. For example, Party V can only generate the second message after having received the first one. This is a good opportunity to implement a variation of the *session types* pattern, which is especially easy in Rust, due to its move semantics. We will demonstrate this on part of the API implementation.

```rust
/// The structure providing all operations for Party V.
pub struct PartyV<S: PartyVState>(S);

// Necessary stuff for session types
pub trait PartyVState {}
impl PartyVState for Msg1Receiver {}
impl PartyVState for Msg2Sender {}
impl PartyVState for Msg3Receiver {}
impl PartyVState for Msg3Verifier {}
```

List. 4.1: `struct`, `trait` and `impl` declarations

In this first excerpt, a structure `PartyV` is defined, which is a tuple struct wrapping a single value `S`, which is generic with the sole requirement that it must implement the trait `PartyVState`. Traits in Rust are like interfaces and typically declare methods that must be implemented, but they can also be empty as in this case and function as *marker traits*. This trait is then "implemented" for several structures, which are defined later on. Since the trait does not declare any functions, these implementation blocks are empty. Thus, these structures are marked as eligible for inclusion in the `PartyV` structure. As the name implies, they are going to serve as the protocol state.

```rust
/// Contains the state to receive the first message.
pub struct Msg1Receiver {
    c_v: Vec<u8>,
    secret: StaticSecret,
    x_v: PublicKey,
    auth: [u8; 64],
    kid: Vec<u8>,
}

impl PartyV<Msg1Receiver> {
    /// Creates a new `PartyV` ready to receive the first message.
    pub fn new(
        c_v: Vec<u8>,
        ecdh_secret: [u8; 32],
        auth_private: &[u8; 32],
        auth_public: &[u8; 32],
        kid: Vec<u8>,
    ) -> PartyV<Msg1Receiver> {
        ...

        PartyV(Msg1Receiver {
            c_v,
            secret,
            x_v,
            auth,
            kid,
        })
    }
```

```
29
30      /// Processes the first message.
31      pub fn handle_message_1(
32          self,
33          msg_1: Vec<u8>,
34      ) -> Result<PartyV<Msg2Sender>, OwnError> {
35          ...
36      }
37  }
```

List. 4.2: State and implementation for first message

In Listing 4.2 we see the definition of one of these state structures, which contains all the initialization data required before starting the protocol run. This is followed by an implementation block for `PartyV` instances that have this initialization data as the state. It contains an associated function that does not make use of this type parameter and is a sort of "constructor", returning a `PartyV` structure that wraps an instance of our `Msg1Receiver` state structure.

The block also has an implementation of the method `handle_message_1`, which takes the bytes of the first message as an argument. It does some processing on this and returns either an error or an instance of `PartyV` wrapping a new state structure, the one required for sending the second message.

```
1  /// Contains the state to build the second message.
2  pub struct Msg2Sender {
3      c_v: Vec<u8>,
4      shared_secret: SharedSecret,
5      x_v: PublicKey,
6      auth: [u8; 64],
7      kid: Vec<u8>,
8      msg_1_seq: Vec<u8>,
9      msg_1: Message1,
10 }
11
12 impl PartyV<Msg2Sender> {
13     /// Returns the bytes of the second message.
14     pub fn generate_message_2(
15         self,
16     ) -> Result<(Vec<u8>, PartyV<Msg3Receiver>), OwnError> {
17         ...
18     }
19 }
```

List. 4.3: State and implementation for second message

Listing 4.3 contains the definition of this state structure, as well as the implementation block for the `PartyV` instance that wraps the structure and is returned from `PartyV::handle_message_1`. We can see that this implementation only applies to `PartyV<Msg2Sender>`, effectively any instance of `PartyV` which has the necessary state to send the second message. Trying to use this method before having handled the first message will result in a compiler error. Additionally, if one were to use an editor with autocompletion, this would be the the only available method it would propose, so the user does not even necessarily have to understand EDHOC or the API in any detail to use it, since they are guided by only being able to do the right thing.

We can also see that the method implicitly takes `self` as an argument, which means it takes ownership of its structure as opposed to merely taking a reference, which would be `&self` or `&mut self` for a mutable one. This is very important, because after the method has been called, the variable holding the instance (the previous owner) is no longer valid, since ownership has been moved. The previous instance has therefore been consumed and a new instance for the next step is returned. Rust's move semantics together with this pattern do not let the user apply the same operation twice when it does not make sense. With these two mechanisms we can prevent misuse of our API at compile time, which is very beneficial to both safety and usability.

4.2. OSCORE

4.2.1. Approach

This implementation was done in a slightly more top-down way than EDHOC, because the API design for OSCORE was more obvious. Since the security context is the foundation of everything in OSCORE, its data structure and derivation were the first building blocks. The functionality concerned with protection and unprotection of messages followed, and just as for EDHOC, lots of independently testable utility functions provide the core functionality for the API abstraction.

It made a big difference that OSCORE—unlike EDHOC—was already a finished RFC (proposed standard since July 2019 [17]) when we started. Not only did this give us some certainty that the specification would not change below our feet, the biggest advantage was that it already came with a good set of test vectors, allowing us to be confident of implementing things the right way from the beginning.

4.2.2. CoAP Integration

There was one aspect that made the OSCORE implementation slightly more challenging than EDHOC—the strong coupling with CoAP as a result of it protecting CoAP messages. As a consequence, this means that an OSCORE implementation needs a way to encode and decode CoAP messages.

One interesting approach to solve this, without having to add CoAP functionality to the library itself, would be to rely on the user to provide a CoAP implementation of their choice through some fixed interface (a set of traits in Rust). In practice, they would have to write a small binding to expose CoAP functionality, such as iterating over the options of a message or accessing the code and payload. The example[5] of this approach that made us consider it was written by somebody working on a C implementation of OSCORE. A big advantage of doing it as proposed, besides not having to concern ourselves with the specifics of CoAP, is that the binary size of projects using it could be reduced. After all, it is very likely that an application using an OSCORE implementation already contains a CoAP library, so reusing it instead of adding another one would reduce code duplication. This is a convincing argument, but while investigating, we noticed something in the OSCORE specification that makes it less enticing. Section 5.3 of [17] specifies that the

[5] `https://gitlab.com/chrysn/coap-message`

plaintext (which will be the payload of the OSCORE message when encrypted) containing the protected code, options and payload, will be formatted like a CoAP message, except for only having a subset of the header. So in practice, an OSCORE implementation needs to be able to construct a CoAP message from scratch and modify it slightly to remove the unnecessary header parts. In essence, the user would therefore have to expose an almost complete set of CoAP functionality, which would be very tedious. But if we look at Figure 3.6 where we can see how the plaintext is formatted, it does not look too complicated. Maybe this CoAP subset could even be implemented directly in our library. Unfortunately, the devil is in the details, the "Class E options" specifically. The CoAP options format is very efficient in order to be as compact as possible. Section 3.1 of [19] explains this in detail. For example, the option numbers are not explicitly included, instead all options are included in ascending order and a delta encoding is used. The result of this is that it is very compact, but easily the most complex part of CoAP encoding. If one were to implement this, they might as well do a full CoAP implementation, which is then just in arm's reach anyway.

Given this situation, we figured the best way forward would be to bite the bullet and rely on a CoAP library directly to handle all of this. While that comes at the cost of increased binary size, it makes everything easier for us and most importantly the user of our library, as they do not have to do anything. The next problem then is finding a suitable library. At the time, we were aware of two almost feasible candidates. `JNeT`[6] is an experimental `#![no_std]` library for many network protocols, among them CoAP. While very competently built, the library is experimental, unsupported and not the most ergonomic to work with. The other was `coap`[7], arguably *the* CoAP library for Rust. While otherwise pretty much perfect, it simply did too much for our purposes. It implements a full CoAP server using UDP and sockets, thus relying on the standard library, so very much not usable in `#![no_std]`. Since neither of them supported the newly introduced `OSCORE` option that we needed, we would have had to fork them anyway. That is why we opted for taking the parts that do CoAP encoding and decoding from the `coap` crate, modifying them so they work in `#![no_std]` and put them in a separate crate[8]. It strikes a good balance between working in `#![no_std]` and still being pleasant to use, by relying on some heap allocation. More about the tradeoffs of that in Section 4.3.

Being able to conveniently work directly with CoAP messages and our continuous verification of individual components against the test vectors, we did not have any other notable issues and everything ended up coming together nicely. We were able to finish the library implementation of EDHOC and OSCORE on schedule, almost exactly eight weeks after the first commit.

4.3. Notes on Heap Allocation

While heap allocation is usually a given in most programming, on bare metal that is not necessarily the case. Because such a `#![no_std]` environment as we call it in Rust is the lowest common denominator of what hardware usually gives us, by default there is no heap allocator that manages and provides this memory for us. This means that types that

[6] `https://github.com/japaric/jnet`
[7] `https://crates.io/crates/coap`
[8] `https://crates.io/crates/coap-lite`

require heap allocation, such as `Vec` or `String` cannot be used in `#![no_std]` by default. Even on systems where heap allocation is possible, it is generally not taken advantage of in embedded programming, because it adds a bit of unpredictability, particularly the need to handle failed allocations in out of memory situations and the fact that memory could become fragmented over time. For systems where heap allocation is possible, Rust allows us to use it with the `alloc` crate. It provides a set of traits and types, which can be used to implement a heap allocator (we will just call it an allocator for brevity) for a specific system, use such an allocator as a global allocator in a binary, as well as declare in a library that it requires a global allocator to be available, which allows it to use types that do heap allocation even in `#![no_std]`. While it would have been possible to implement everything in EDHOC and OSCORE just with stack-allocated fixed-size buffers, it would have been less ergonomic and taken more time, especially since some useful features of libraries we use (e.g. CBOR encoding) require allocation themselves. Since an allocator is available for the Cortex-M platform, which we targeted for the demo, and the `alloc` crate was conveniently stabilized in Rust 1.36.0 just before we started [22], we decided to accept the tradeoffs of using the heap for some operations that can benefit from it. This has enabled us to be more productive and in our testing we have not observed any adverse effects, such as memory fragmentation. A continuous exchange of OSCORE messages over two hours did not show any issues. This is most likely due to the fact that only a small select portion of the overall memory usage is on the heap and that practically all of the values allocated on the heap while processing a message are deallocated straight after. While it is possible to leak memory even in Rust in very special circumstances (using reference counted types for example), our implementation does not use any of the features that allow this to happen and is therefore completely free of memory leaks.

The timely stabilization of `alloc` was quite fortunate, because unstable features require a nightly compiler. Rust makes it very easy to manage and use stable, beta and nightly toolchains side by side, so that is not usually a problem for binaries like the demo we built. But making a library that requires a nightly compiler is inadvisable, because it tends to be infectious, forcing all libraries and binaries that use it as a dependency to do the same. The only difficulty we faced was that a small crate[9], which is responsible for an aspect of CBOR serialization and uses allocation for a subset of its functionality, was not updated to take advantage of the stabilization. It still required the unstable allocation feature, which would have made our library do the same. Because this was the only crate holding us back, we pushed a small pull request to fix it and now our library compiles with stable Rust, a benefit that was certainly worth the little time we spent on getting this fixed.

[9] `https://github.com/serde-rs/bytes`

5 Results

5.1. Overview

The main body of work is of course the combined EDHOC and OSCORE library[1], and to a lesser extent also the two standalone libraries for AES-CCM[2] and CoAP[3]. In order to be certain that it is suitable for actual use, we also built a demonstration setup[4] for it, which is presented in the following section.

5.2. Experiment

To demonstrate our library working on real hardware, we decided to implement a resource server on a STM32F303VCT6 with a client on a STM32F407VGT6U, which are Class 2 constrained nodes [3]. These devices are well-supported and it is very easy to get started programming for them using a quickstart template[5].

[1] `https://github.com/martindisch/oscore`
[2] `https://github.com/martindisch/aes-ccm`
[3] `https://github.com/martindisch/coap-lite`
[4] `https://github.com/martindisch/oscore-demo`
[5] `https://github.com/rust-embedded/cortex-m-quickstart`

It was not all smooth sailing though. We encountered the same jarringly "weird" issues people tend to get hung up on when stepping down from metaphorical software heaven into hardware hell. Some examples to give the reader a taste:

- A STM32VLDISCOVERY board did not seem to connect via Universal Serial Bus (USB). It presents itself to the host as a composite USB device with the programmer interface, as well as a USB mass storage device with some informational documents. The firmware implementation of that mass storage device was buggy, causing repeated resets, thus making the device unavailable. Forcing the host to ignore the mass storage device fixed the problem.
- The Instrumentation Trace Macrocell (ITM) is a core peripheral available on some microcontrollers that allows for faster and more efficient logging than serial interfaces. There is an interesting in-depth article[6] about this topic in the context of Rust embedded programming. For some reason, there were issues with it on the F3, forcing us to resort to simply logging over Universal Synchronous/Asynchronous Receiver/Transmitter (USART), which was more than sufficient for the demo.
- When trying to use the F4 in standalone mode—that is without a host computer and simply powered over USB from a regular power supply—it did not do anything at all. This time, it was also a bug in the ST-Link firmware, which made the on-device debugger hold the user program in reset, unless it was connected to a host computer[7]. A firmware update fixed this.

But the biggest problem we had was that of Ethernet connectivity. Initially, we decided to work with the Waveshare ENC28J60 Ethernet board[8], because there already was a Rust driver for working with the ENC28J60 controller[9]. We were able to do some experiments with it and it seemed to mostly work. However, as our demo progressed, we noticed that it was still strangely unreliable. There were several separate issues, which we have to go into in order.

First of all, there were cases where packets received in a previous run were received again after restarting the program. This suggests that the internal reception buffer of the Ethernet controller was not reset properly. The datasheet[10] mentions that for a successful reset, the reset pin has to be held low for at least 400 ns, otherwise the change in signal will be filtered out. And indeed, the driver did not ensure this was the case, but simply set the pin to low and high again with the next instruction. Maybe the developer tested the driver on a microcontroller that runs at sufficiently low frequency such that 400 ns passed between the two invocations. We were able to verify that introducing a delay did solve the problem, but in the end decided to use a Serial Peripheral Interface (SPI) command for resetting, because that worked just as well and did not require an extra wire.

Much more serious were the seemingly random panics (unrecoverable errors, resulting in a crash) of the driver. We traced these back to unhandled error conditions. Modifying the driver to make sure these errors did not cause a panic but returned a proper error instead, allowed our application to handle them by retrying the operation. Still, the underlying problems of the strange errors remained. The ENC28J60 writes 64 respectively 32 bit

[6] `https://blog.japaric.io/itm/`
[7] `https://os.mbed.com/questions/68969/STM32F429I-DISC1-can-not-be-powered-by-u/`
[8] `https://www.waveshare.com/enc28j60-ethernet-board.htm`
[9] `https://crates.io/crates/enc28j60`
[10] `https://www.waveshare.com/w/upload/7/7f/ENC28J60.pdf`

status vectors after transmitting and receiving, which the driver reads. These can indicate a number of error conditions, such as packets that are longer than Ethernet frames are allowed to be. But the values we received in these status vectors did not make much sense. Often it indicated packets of an enormous length that we were sure were not actually sent, other times the status vector was completely empty (just zeros) or displayed an unlikely combination of flags that were nonsensical, as if we were reading a random area of memory. Besides a hardware issue, our best guesses are that either there is a deeper problem in the driver implementation, or more likely that there are subtle differences between our board that hosts the Ethernet controller and that of the driver author, which somehow change how the ENC28J60 controller operates. That should not be the case, but it is imaginable. We tried to make it work with our modified driver, by catching errors and retrying on failure. That worked well enough for errors during sending, but resetting the device on an error while receiving caused message loss, which was unacceptable. We therefore decided to look into the WIZnet WIZ850io[11], for which there is also a Rust driver[12]. While the API of this driver is not documented as well and more cumbersome, the W5500 Ethernet controller used on this board offers more features than the ENC28J60 and is therefore more pleasant to use. It has proven completely reliable and we have not had any problems with it since.

The next step was verifying that our implementation worked as expected. We started by creating the resource server on the F3 and testing it with a desktop client. After porting the client to the F4 and seeing it work in conjunction with the server, the basic requirements were fulfilled. But because it was straightforward and a good demonstration of how OSCORE allows for proxying, we decided to throw a proxy in the mix. We implemented a very naive CoAP proxy in 60 lines of Rust code that ran on a Raspberry Pi and simply forwarded CoAP packets to the requested destination.

[11] `https://www.wiznet.io/product-item/wiz850io/`
[12] `https://crates.io/crates/w5500`

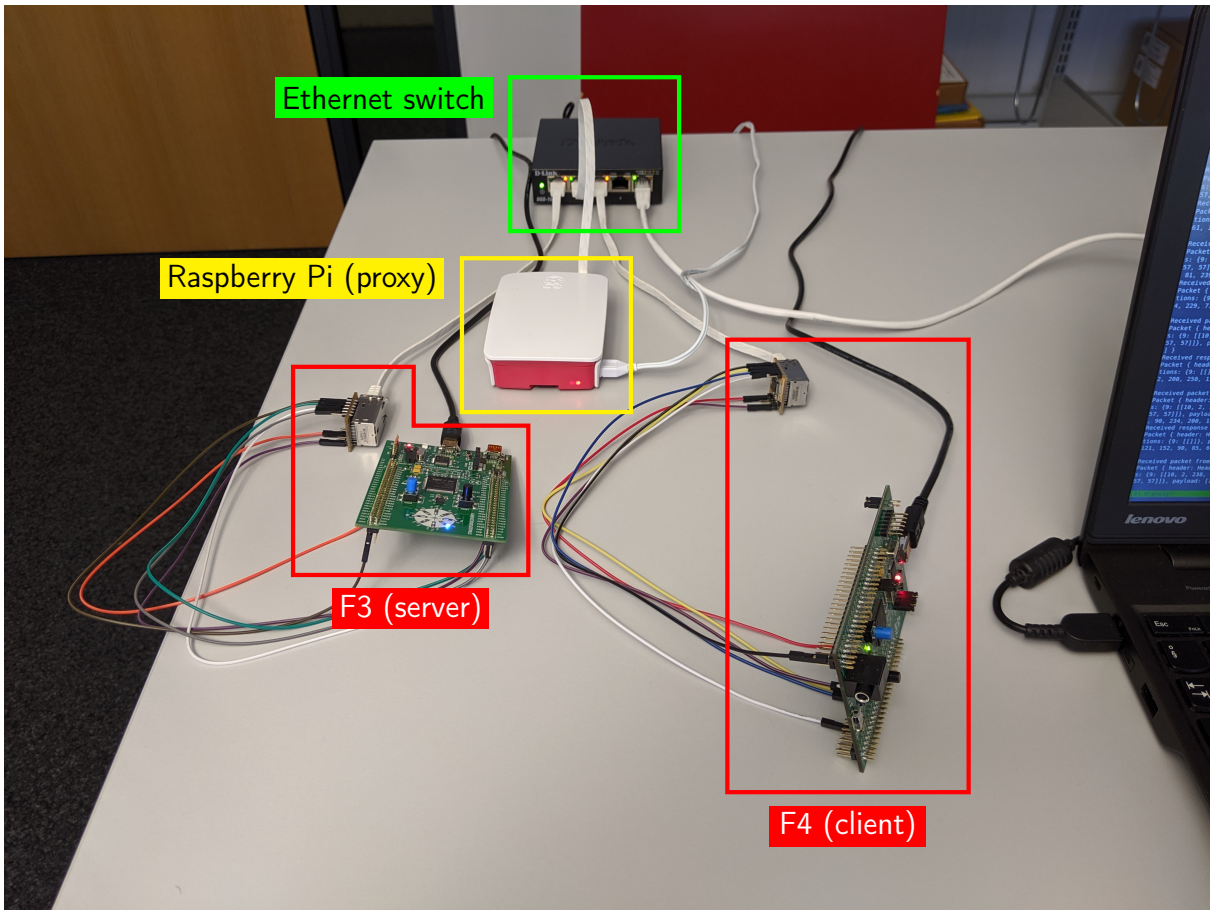

Fig. 5.1.: Demo setup

This completed our demo. The CoAP server provides two dummy resources: `/hello` always returns a "hello world" type message and `/echo` responds with the payload of the request. While it was quickly put together for the demo and is far from a reference implementation of a reusable CoAP server, it correctly implements the subset of functionality it needs and can be used with any CoAP client. The same goes for the proxy, which can also be used with an arbitrary CoAP client to route requests and responses to and from the server. The computer is only connected to the switch for a Secure Shell (SSH) connection to the proxy to show its output. As soon as it is connected to power, the client initiates the EDHOC exchange with the server, after which it will start an endless loop of making CoAP requests protected with OSCORE to the server, one to the `/hello` resource, then one to the `/echo` resource and so on. The requests to `/echo` include a counter to easily keep track of how many messages have already been exchanged.

There is also a short video[13] of an earlier test of the demo. In this instance, the left terminal on the computer shows the messages that pass through the proxy and the right one the debug output from the client, which is received via serial connection.

The demo also allows us to investigate the cost of using our library a bit better, by looking at several metrics we have collected on the embedded resource server. The benefit of looking at not just the library itself, but a binary using it, is that we can use the great tools we have available for the target platform to do so. The disadvantage is, of course, that the results will be distorted somewhat by the additional cost of the application

[13] `https://photos.app.goo.gl/8TWcciupGW8YoXt48`

code, libraries it uses such as the HAL implementation or the driver for the Ethernet module, or the minimal runtime that is included in the binary for things like formatting panic messages. Still, the result shows us accurately what a typical embedded application using our library would look like, since that is exactly what our resource server is. In the following sections where we go into several different metrics, it will be made clear whether the presented information pertains to just the library itself, or the embedded resource server using it.

5.3. Code Analysis

While the library contains both EDHOC and OSCORE, they are not coupled in any way, but exist as completely independent modules within them. Each can easily be used on its own. Due to the very careful approach of testing every unit of code as early as possible, the overall test coverage is at over 95%.

```
$ cargo tarpaulin
[INFO tarpaulin] Running Tarpaulin
[INFO tarpaulin] Building project
[INFO tarpaulin] Launching test
[INFO tarpaulin] running /home/martin/Projects/oscore/target/
                          debug/deps/oscore-cecde91908de6476

running 51 tests
...................................................
test result: ok. 51 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

[INFO tarpaulin] Coverage Results:
|| Tested/Total Lines:
|| src/cbor/mod.rs: 60/60
|| src/edhoc/api.rs: 327/340
|| src/edhoc/cose.rs: 95/100
|| src/edhoc/util.rs: 271/280
|| src/lib.rs: 6/6
|| src/oscore/context.rs: 263/282
|| src/oscore/util.rs: 306/320
||
95.68% coverage, 1328/1388 lines covered
```

List. 5.1: Test run

The number of lines shown in the tool does not accurately reflect the actual number of lines in the source files, but what it sees as discrete lines, which can in fact span multiple lines.

A more classical analysis of the source files reveals that we have in fact about 3200 lines of code in the whole project. Almost exactly a third of these are unit tests and a few more are example code, which explains the substantial difference.

Language	Files	Lines	Blank	Comment	Code
Rust	16	4465	422	834	3209
Toml	1	38	3	2	33
Markdown	1	41	10	0	31
Total	18	4544	435	836	3273

Tab. 5.1.: File stats of the complete repository

Removing tests and examples, we end up with a more accurate picture which reveals that about 25% of non-blank lines are comments and documentation.

Language	Files	Lines	Blank	Comment	Code
Rust	14	3078	270	752	2056
Total	14	3078	270	752	2056

Tab. 5.2.: File stats for source without tests

It was very important to us that we properly document everything for ourselves, as well as others, and we have been rewarded for this policy more than once when revisiting code we had written a number of weeks earlier. The generated documentation for the library is published on GitHub[14].

5.4. Binary Size

Looking at the different segments of the embedded resource server binary can show us how much flash size it will occupy.

```
$ cargo size --bin server --release -- -A
server  :
section               size        addr
.vector_table          404   0x8000000
.text                110514   0x8000194
.rodata               51940   0x801b150
.data                     0  0x20000000
.bss                     20  0x20000000
.uninit                   0  0x20000014
.debug_str           511082         0x0
.debug_loc           368359         0x0
.debug_abbrev          3637         0x0
.debug_info          553733         0x0
.debug_ranges        116048         0x0
.debug_macinfo            2         0x0
.debug_pubnames      173957         0x0
.debug_pubtypes      174006         0x0
.ARM.attributes          56         0x0
.debug_frame          18808         0x0
.debug_line          168838         0x0
.debug_aranges          296         0x0
.comment                175         0x0
```

[14] https://martindisch.github.io/oscore/oscore/

```
23 Total              2251875
```

List. 5.2: Segments of server binary

Since only the `.vector_table`, `.text` and `.rodata` segments will occupy space in the flash memory (as indicated by the addresses), the flash size of this binary is 162858 bytes.

We were a bit surprised at the size, which could have been smaller. There is a tool[15] to investigate which parts of the code take up the most space, but unfortunately it only supports ELF (Linux, BSD), Mach-O (macOS) and PE (Windows) binaries. However, we quickly suspected the CBOR implementation as the culprit, because it uses a very popular generic serialization/deserialization framework for Rust. It makes heavy use of *static dispatch* (instead of *dynamic dispatch*) through a technique called *monomorphization*, which is usually great, because it makes generics zero-cost and allows for some optimizations. The disadvantage is that it can significantly increase binary size. For more on this interesting topic, read about monomorphization[16] and trait objects[17] in Rust. Since we could not use a tool to inspect our binary, we resorted to making CBOR serialization/deserialization calls in our code do nothing, effectively removing them. The resource server with this version weighed only 98818 bytes, a reduction of almost 40%. This confirmed our suspicion, replacing the CBOR handling with something more lightweight would therefore be the first step in improving the binary size.

5.5. Memory Usage

The `call-stack` tool[18] can be used to determine memory usage, by statically analyzing the generated LLVM Intermediate Representation (IR) of an application. We applied this to the embedded resource server. It creates the full call stack as a DOT file (graph representation format), which shows all possible execution paths, as well as the inferred local stack usage of every function. By adding up that and the local stack usage of every function it calls, as well as the callees of those and so on, it can also determine the full stack usage caused by calling the function. Of course that is no longer possible if there are cycles in the call graph (e.g. from recursion), as it cannot know how many levels deep such a cycle will go at runtime. It will then simply display the minimum stack usage. Since that is not typically the case, we can safely assume the values to be the accurate cost accrued by using a function. There is also another caveat—inlining. Because the tool works on the IR, many optimizations have already been made, which is useful, since it more accurately reflects the runtime behavior. But it also means that some functions may have been inlined into others, "disappearing" in the process and adding their cost to the local usage of the parent function, which makes it hard to discover the true source of some memory usage. In the application, this can be prevented by annotating functions with `#[inline(never)]` to tell the compiler not to inline them, but of course that has no effect on the libraries we use. Since we are optimizing our binary for size and not speed, inlining is kept to a minimum anyway and the call graph we get is fairly accurate.

[15] `https://crates.io/crates/cargo-bloat`

[16] `https://doc.rust-lang.org/stable/book/ch10-01-syntax.html#performance-of-code-using-generics`

[17] `https://doc.rust-lang.org/stable/book/ch17-02-trait-objects.html#trait-objects-perform-dynamic-dispatch`

[18] `https://crates.io/crates/cargo-call-stack`

There are several viewers for DOT files and a very simple way is to convert it to a Scalable Vector Graphics (SVG) file. However, this is not really practical due to the size of the call graph of the resource server. Most viewers crash trying to render the SVG. And for good reason, since the call graph is 2.2 meters wide and 0.75 meters high at default font size.

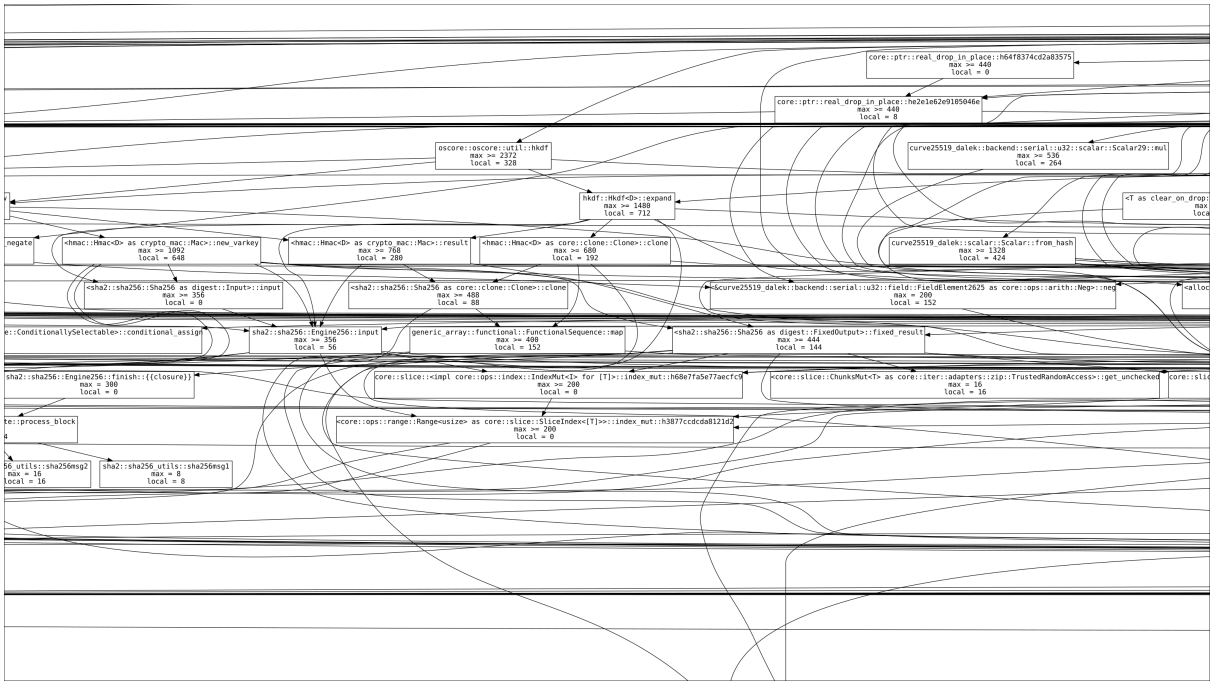

Fig. 5.2.: Call graph excerpt

Most of the complexity comes from implementation details of the language, its core library or other libraries, and is not particularly relevant to us, so as one might be able to guess from Figure 5.2, it is difficult to analyze it meaningfully. To do so anyway, we wrote a small tool to parse the DOT file, present us with a list of function calls for a specific function ordered by their accumulated stack usage, and allow us to conveniently traverse the relevant parts of the call graph like that.

```
1      22848   3000    main
2
3  0:  19848   3888    server::oscore::OscoreHandler::handle
4  1:    392     32    alloc::slice::<impl [T]>::to_vec::h75fcf83c4bab5683
5  2:    272      0    alloc::raw_vec::RawVec<T,A>::dealloc_buffer::h0f3f40e8a0a5cb9a
6  3:    256      0    core::result::Result<T,E>::expect::hf0f151e14425c943
7  ...
```

List. 5.3: `main` function

This is part of the output it gives us for the top-level `main` function. It tells us that the function uses 3000 bytes locally and with all the functions it calls, it comes in at about 22 KiB at its peak, which leaves more than enough room in the 40 KiB Random-Access Memory (RAM) of the F3 for the small amount of heap it needs. Below that we can see the top functions it calls, with their total and local stack usage. Of course the `OscoreHandler` takes up most of the RAM, since everything passes through it first, before being delegated to the CoAP and EDHOC handling.

```
    19848   3888    server::oscore::OscoreHandler::handle

0:  15960    192    server::coap::CoapHandler::handle
1:   8448   6904    <aes_ccm::ccm::AesCcm<TagSize> as aead::NewAead>::new
2:   2380    328    oscore::oscore::util::hkdf
3:    688    184    coap_lite::packet::Packet::from_bytes
4:    664     64    coap_lite::packet::Packet::add_option
...
```

List. 5.4: `OscoreHandler::handle` function

Taking a look at the `OscoreHandler`, we see that its call to the `CoapHandler` is the most expensive, as expected. Surprisingly, the AEAD it uses for encryption—AES-CCM—is a close second. This can be explained, since AES implementations are known to allocate large tables for speedups.

```
     8448   6904    <aes_ccm::ccm::AesCcm<TagSize> as aead::NewAead>::new

0:   1544   1320    aes_soft::bitslice::bit_slice_fill_4x4_with_u32x4
1:    224    192    <generic_array::GenericArray<T,N> as generic_array::sequence::
                    GenericSequence<T>>::generate::h08f91fac028cb4f7
2:    200     48    core::panicking::panic_bounds_check
...
```

List. 5.5: AES functionality

Investigating AES-CCM a bit more closely, we can see the issue of inlining in action. AES-CCM is provided by a library, so without modifying it to litter `#[inline(never)]` everywhere, functions will be inlined where it makes sense. This seems to have been the case here, as the `CcmMode::new` function uses 6904 bytes locally of a total usage of 8448 bytes. Since we wrote the library ourselves, we happen to know that this constructor does not do much besides initializing the underlying AES implementation, so most of the stack usage should be attributed to that, not our constructor. But the corresponding function call is ominously missing, indicating that it has been inlined and its stack usage added to that of the constructor function. After all, the most demanding function call we get in the list is a (relatively) small one from the AES implementation to build some matrices. Most likely, the initialization has been inlined since it was only called once, in which case inlining does not result in a larger code size, whereas the `bit_slice_fill_4x4_with_u32x4` function is called more frequently, so inlining would be against our optimization goal (small size).

For now, we go back up one level and down into the `CoapHandler` to continue the investigation.

```
    15960    192    server::coap::CoapHandler::handle

0:  15768   7320    server::edhoc::EdhocHandler::handle
1:    784     88    server::coap::generate_response
2:    776     80    server::coap::generate_link_format
...
```

List. 5.6: `CoapHandler::handle` function

Once more, we are referred to the next component, the `EdhocHandler` in this instance. We can also see that the actual handling of the CoAP messages is relatively inexpensive,

although we do not get the full picture here, since this is one of the few places where a bit of heap allocation happens that is hidden in static analysis.

```
     15768   7320    server::edhoc::EdhocHandler::handle

 0:   8448   6904    <aes_ccm::ccm::AesCcm<TagSize> as aead::NewAead>::new
 1:   2468    416    oscore::edhoc::util::edhoc_key_derivation
 2:   2016    592    server::edhoc::EdhocHandler::initialize
 3:   1484    176    curve25519_dalek::edwards::EdwardsPoint::compress
 ...
```

List. 5.7: `EdhocHandler::handle` function

Inside the `EdhocHandler`, we have come to an end with nothing new to discover. We find out that most of the memory is again used by AES, followed by other cryptographic operations like key derivation or preparing Curve25519, used for both ECDH and signatures.

In conclusion, the most straightforward way to reduce the memory requirements would be to use a cryptographic element in hardware, which would of course have to support all the ciphers we need. Considering that AES is the main offender, having one that does just that might be enough to considerably reduce memory pressure. Some microcontrollers come with AES support built in, which would be even better. Another approach would be using a different software implementation, akin to TinyCrypt, which is optimized for size at the cost of speed.

5.6. Benchmarks

Two kinds of benchmarks were used in this project. The purpose of the first type was to get a sense of the relative cost of operations (how they compare to each other) and to support development. These benchmarks are part of the the library project and were built with Criterion.rs[19], a statistics-driven microbenchmarking suite for Rust, which integrates nicely with the Cargo build system.

```
$ cargo bench
edhoc_detailed/party_u_build
                        time:   [32.403 us 32.405 us 32.408 us]
Found 4 outliers among 100 measurements (4.00%)
  2 (2.00%) high mild
  2 (2.00%) high severe
edhoc_detailed/msg1_generate
                        time:   [410.51 ns 414.38 ns 418.29 ns]
Found 16 outliers among 100 measurements (16.00%)
  2 (2.00%) low severe
  3 (3.00%) low mild
  7 (7.00%) high mild
  4 (4.00%) high severe
...
```

List. 5.8: Benchmark usage

These results are very accurate, reproducible, and correctly show the relative amounts of time spent in different parts of the overall execution. Implementing benchmarks in this

[19] `https://github.com/bheisler/criterion.rs`

manner is extremely beneficial for development, as it enables comparing the performance of changes with that of the previous version.

```
edhoc_detailed/msg1_generate
                        time:   [485.83 ns 503.35 ns 522.53 ns]
                        change: [+12.251% +15.298% +18.419%] (p = 0.00 < 0.05)
                        Performance has regressed.
```

List. 5.9: Performance regression example

While this information is very useful, it does not convey a sense of how the code performs in the real world on embedded devices. To find out, we ran a second type of benchmark on the F3. The cycle counter from the Debug Watch and Trace (DWT) module found on Cortex-M processors was used to measure the number of CPU cycles for operations.

In the following two sections we will discuss the results for EDHOC and OSCORE. For each, the desktop benchmarks are presented first to discuss how they compare to each other, afterwards the cycle counts and corresponding execution times at different clock speeds on embedded devices are shown. The full benchmark results with more information about the environments they were produced in are available in Appendix A.

5.6.1. EDHOC

Table 5.3 lists all functions of the public API in the order they are executed in an exchange, as well as the most important operations they entail.

Party	Function	Details	Time [µs]
U	`new`	Struct initialization with authentication and ephemeral key	32.405
	`generate_message_1`	-	0.414
V	`new`	Struct initialization with authentication and ephemeral key	32.438
	`handle_message_1`	Shared secret derivation	98.699
	`generate_message_2`	Hash, signature, key derivation, AEAD encryption	63.062
U	`extract_peer_kid`	Shared secret derivation, hash, key derivation, AEAD decryption	126.030
	`verify_message_2`	Signature verification	108.530
	`generate_message_3`	Hash, signature, key derivation, AEAD encryption, OSCORE context derivation	70.626
V	`extract_peer_kid`	Hash, key derivation, AEAD decryption	27.345
	`verify_message_3`	Signature verification, OSCORE context derivation	116.720

Tab. 5.3.: EDHOC benchmarks (desktop)

As expected, cryptographic operations seem to account for most of the execution time. For example, generating the first message (which does not involve any of those at all) is finished virtually instantly.

Summing up these values results in a total execution time of 338.005 µs for Party U and 338.264 µs for Party V. This is easily verified by a second set of benchmarks, which measure all the operations of each party combined. Indeed, their results are 339.89 µs for Party U and 339.85 µs for Party V, the small difference being a testament to the accuracy of the benchmarks.

On the embedded side, we measure CPU cycles. By default, the F3 and F4 use their internal RC oscillators as clock sources, which run at 8 MHz and 16 MHz respectively. Using an external clock source and a Phase Locked Loop (PLL) to multiply this clock speed, they can go as fast as 72 MHz and 168 MHz. Table 5.4 shows the CPU cycles and corresponding execution times at different frequencies.

Party	Function	CPU cycles	8 MHz [ms]	72 MHz [ms]	168 MHz [ms]
U	`new`	1128152	141.0	15.7	6.7
	`generate_message_1`	9309	1.2	0.1	0.1
V	`new`	1128145	141.0	15.7	6.7
	`handle_message_1`	3198029	399.8	44.4	19
	`generate_message_2`	1920170	240.0	26.7	11.4
U	`extract_peer_kid`	3818507	477.3	53	22.7
	`verify_message_2`	3306214	413.3	45.9	19.7
	`generate_message_3`	2163710	270.5	30.1	12.9
V	`extract_peer_kid`	626818	78.4	8.7	3.7
	`verify_message_3`	3552819	444.1	49.3	21.1

Tab. 5.4.: EDHOC benchmarks (embedded)

Relative to each other, these results are remarkably similar to the desktop benchmarks. Summing up the CPU cycles gives a total of 10425892 for Party U and 10425981 for Party V. This can be verified too, as benchmarks over all operations of each party give 10428827 for Party U and 10428274 for Party V. Overall, every party spends about 1.304 s at 8 MHz, 145 ms at 72 MHz, or 62 ms at 168 MHz.

5.6.2. OSCORE

For OSCORE, there are fewer benchmarking opportunities, as the API is very simple. There is initialization, protecting either a request or a response and unprotecting a request or a response. Which of those it is has little effect on the speed of the operation.

Operation	Time [µs]
Context construction	9.0540
Request protection	11.023
Request unprotection	11.053

Tab. 5.5.: OSCORE benchmarks (desktop)

OSCORE operations are much less demanding than EDHOC, since they do significantly less cryptography. No ECDH, EdDSA, and HKDF (except in context construction). Only AEAD, which is quite fast.

The embedded benchmarks tell a similar story, although here we see a slight divergence between the desktop and embedded versions. In the former, context construction is slightly faster than protection and unprotection, while in the latter they are all roughly the same.

Operation	CPU cycles	8 MHz [ms]	72 MHz [ms]	168 MHz [ms]
Context construction	294072	36.8	4.1	1.8
Request protection	295206	36.9	4.1	1.8
Request unprotection	297662	37.2	4.1	1.8

Tab. 5.6.: OSCORE benchmarks (embedded)

5.7. Limitations

As we mentioned in Section 3.4.1, we implemented one of three authentication mechanisms for EDHOC. We also omitted some aspects that were irrelevant for our application, such as the optional application data that can be transported in EDHOC messages and the negotiation of a cipher suite, since we only implemented the mandatory one.

For OSCORE, we implemented one of three mechanisms for deriving the security context, as mentioned in Section 3.5.2. We did not implement some of the optional features, such as handling of the special options for `Observe` or block-wise transfer.

All of these were omissions due to time constraints. It would have been feasible to do a full implementation of either EDHOC or OSCORE, but we were quite challenged by Learning Rust, embedded programming, understanding the two main and several related RFCs, implementing the main libraries and the additional ones, as well as standing up the demo. Despite that, we took great care to ensure that everything we did implement complies with the test vectors and is correct, so that it can serve as the base for a full implementation.

The main limitation of the demo is that the F3 does not have a True Random Number Generator (TRNG). Without this, it is not possible to generate randomness that is cryptographically secure, and therefore not possible to create the required ephemeral keys for the ECDH exchange. The F4 does have one, but only one party being able to generate secure ephemeral keys does not really make the exchange as a whole more secure. It is important to state though that this is only a limitation of the demo and not of the library. The library is I/O-free and does not deal in generating randomness. That is the task of the user, who can do it with whatever method is most appropriate for their hardware. All the library needs from the user then are 32 truly random bytes, which are used in the ECDH exchange. We therefore decided to simply use the “ephemeral” keys from the test vectors in the demo, which does not detract from the demo as a test case for the library, since generating randomness is not part of its responsibility in the first place.

6
Future Work

Given enough time, there are several interesting opportunities to improve or extend the implementation:

- OSCORE is designed to work well in constrained networks, due to its small message size [17]. The EDHOC draft specifically mentions its suitability for "network technologies such as Cellular IoT, 6TiSCH, LoRaWAN" [16]. It would therefore be interesting to demonstrate our implementations using these wireless technologies.
- Implement additional mechanisms and optional features for EDHOC and OSCORE to move towards full implementations.
- Update the EDHOC implementation with new symbol names (some were changed in the latest draft, such as `X_U` to `G_X`) to make the code more easily readable next to the current draft and apply some improvements based on the experience we gained.
- Add support for more than just the mandatory cipher suites.
- Stack-allocate everything, so instead of little there is no use of the heap at all.
- Use a different software implementation of AES. The current one is good, but it is built for speed and therefore does some optimizations at the cost of higher memory usage. For our purposes, something like TinyCrypt that is optimized for size (binary and memory) would be more suitable.
- Use a purpose-built CBOR implementation. While the crate we use now is very well made, it is a bit tricky to use in our situation where we could do with a simpler solution. It does not support CBOR sequences yet, which results in more code being written by the consumer, as well as bigger risk of introducing bugs. Additionally, as we found out in Section 5.4, it takes up a lot of space in the binary.
- Contribute to the W5500 driver for a more convenient API and better documentation.

7 Conclusion

In this project we have implemented standards for securing communication between constrained devices from two RFCs as an open source library and demonstrated it working on real hardware with a resource server, proxy and client. As a byproduct, two smaller libraries for CoAP handling and AES-CCM were also created.

While the main appeal of the project initially was that it would allow us to learn and apply Rust, which we had wanted to do for a while, we came to really appreciate EDHOC and OSCORE as the neat solutions to real problems that they are. Being able to work with the authors of EDHOC to a small extent was an unexpected highlight of the project, as was porting TinyCrypt's CCM mode implementation, which was a first for us and a really nice experience. Over the months we have spent working on this, there have been more of these moments than we could possibly list here. From these small successes, to seeing the first EDHOC exchange complete, or two embedded devices communicate with each other with OSCORE for the first time.

It was a project full of firsts for us—learning Rust, delving into embedded development, implementing an RFC from scratch or porting a cryptographic algorithm. At the beginning, we were not sure we would be able to do all of this in the time we had. But it worked out and from this experience we can only recommend taking a leap of faith every now and then as an opportunity to grow.

A
Benchmarks

The desktop benchmarks were run on a system running Arch Linux with kernel 5.3.13 on an Intel i7-6700K (8) @ 4.200GHz CPU, with the Rust compiler rustc 1.39.0 (4560ea788 2019-11-04). For EDHOC, the data from the test vector for asymmetric authentication was used. For OSCORE, it was test vector 1 for context construction and test vector 4 for protection/unprotection.

```
edhoc_detailed/party_u_build
                        time:   [32.403 us 32.405 us 32.408 us]
Found 4 outliers among 100 measurements (4.00%)
  2 (2.00%) high mild
  2 (2.00%) high severe
edhoc_detailed/msg1_generate
                        time:   [410.51 ns 414.38 ns 418.29 ns]
Found 16 outliers among 100 measurements (16.00%)
  2 (2.00%) low severe
  3 (3.00%) low mild
  7 (7.00%) high mild
  4 (4.00%) high severe
edhoc_detailed/party_v_build
                        time:   [32.426 us 32.438 us 32.457 us]
Found 5 outliers among 100 measurements (5.00%)
  2 (2.00%) high mild
  3 (3.00%) high severe
edhoc_detailed/msg1_handle
                        time:   [98.690 us 98.699 us 98.709 us]
Found 11 outliers among 100 measurements (11.00%)
  1 (1.00%) low mild
  5 (5.00%) high mild
  5 (5.00%) high severe
edhoc_detailed/msg2_generate
                        time:   [63.039 us 63.062 us 63.085 us]
Found 2 outliers among 100 measurements (2.00%)
  2 (2.00%) high mild
edhoc_detailed/msg2_extract
                        time:   [126.02 us 126.03 us 126.05 us]
Found 4 outliers among 100 measurements (4.00%)
  1 (1.00%) low mild
  1 (1.00%) high mild
  2 (2.00%) high severe
edhoc_detailed/msg2_verify
                        time:   [108.52 us 108.53 us 108.55 us]
```

```
Found 5 outliers among 100 measurements (5.00%)
  2 (2.00%) high mild
  3 (3.00%) high severe
edhoc_detailed/msg3_generate
                        time:   [70.609 us 70.626 us 70.644 us]
Found 8 outliers among 100 measurements (8.00%)
  6 (6.00%) high mild
  2 (2.00%) high severe
edhoc_detailed/msg3_extract
                        time:   [27.320 us 27.345 us 27.374 us]
Found 6 outliers among 100 measurements (6.00%)
  2 (2.00%) high mild
  4 (4.00%) high severe
edhoc_detailed/msg3_verify
                        time:   [116.68 us 116.72 us 116.80 us]
Found 7 outliers among 100 measurements (7.00%)
  1 (1.00%) high mild
  6 (6.00%) high severe

edhoc_full/party_u      time:   [339.89 us 339.96 us 340.06 us]
Found 10 outliers among 100 measurements (10.00%)
  4 (4.00%) low mild
  2 (2.00%) high mild
  4 (4.00%) high severe
edhoc_full/party_v      time:   [339.85 us 339.92 us 340.00 us]
Found 10 outliers among 100 measurements (10.00%)
  4 (4.00%) high mild
  6 (6.00%) high severe

oscore/context_derivation
                        time:   [9.0526 us 9.0540 us 9.0555 us]
Found 11 outliers among 100 measurements (11.00%)
  5 (5.00%) low severe
  1 (1.00%) high mild
  5 (5.00%) high severe
oscore/protection_request
                        time:   [11.019 us 11.023 us 11.029 us]
Found 13 outliers among 100 measurements (13.00%)
  2 (2.00%) low severe
  7 (7.00%) low mild
  2 (2.00%) high mild
  2 (2.00%) high severe
oscore/unprotection_request
                        time:   [11.039 us 11.053 us 11.076 us]
Found 1 outliers among 100 measurements (1.00%)
  1 (1.00%) high severe
```

List. A.1: Desktop benchmark output

The embedded benchmarks were run on an STM32F303VCT6 at the default clock of 8 MHz (although that is irrelevant, since it measures CPU cycles, not execution time). The cycle counter from the DWT module was used to count CPU cycles. The contents of the functions are the same as in the desktop benchmarks.

```
Basic initialization done
edhoc_detailed::party_u_build took 1128152 CPU cycles
edhoc_detailed::msg1_generate took 9309 CPU cycles
edhoc_detailed::party_v_build took 1128145 CPU cycles
edhoc_detailed::msg1_handle took 3198029 CPU cycles
edhoc_detailed::msg2_generate took 1920170 CPU cycles
edhoc_detailed::msg2_extract took 3818507 CPU cycles
edhoc_detailed::msg2_verify took 3306214 CPU cycles
edhoc_detailed::msg3_generate took 2163710 CPU cycles
edhoc_detailed::msg3_extract took 626818 CPU cycles
edhoc_detailed::msg3_verify took 3552819 CPU cycles
edhoc_full::party_u took 10428827 CPU cycles
edhoc_full::party_v took 10428274 CPU cycles
oscore::context_derivation took 294072 CPU cycles
oscore::protection_request took 295206 CPU cycles
oscore::unprotection_request took 297662 CPU cycles
Benchmarks finished
```

List. A.2: Embedded benchmark output

B
Common Acronyms

6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
6TiSCH	IPv6 over the TSCH mode of IEEE 802.15.4e
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
API	Application Programming Interface
AMR	Automatic Meter Reading
CBOR	Concise Binary Object Representation
CCM	Counter with CBC-MAC
COSE	CBOR Object Signing and Encryption
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial-of-Service
DWT	Debug Watch and Trace
DoS	Denial-of-Service
DTLS	Datagram Transport Layer Security
ECDH	Elliptic Curve Diffie-Hellman
EdDSA	Edwards-Curve Digital Signature Algorithm
EDHOC	Ephemeral Diffie-Hellman Over COSE
HAL	Hardware Abstraction Layer
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IR	Intermediate Representation
ITM	Instrumentation Trace Macrocell
JOSE	Javascript Object Signing and Encryption
JSON	JavaScript Object Notation
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
LPWAN	Low Power Wide Area Networks
LR-WPAN	Low-Rate Wireless Personal Area Networks
MAC	Media Access Control
NB-IoT	Narrowband Internet of Things
OSCORE	Object Security for Constrained RESTful Environments
PLL	Phase Locked Loop

PSK	Pre-Shared Keys
RAII	Resource Acquisition Is Initialization
RAM	Random-Access Memory
RC	Resistor–Capacitor
REST	Representational State Transfer
RFC	Request for Comments
RPK	Raw Public Keys
SPI	Serial Peripheral Interface
SSH	Secure Shell
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
TRNG	True Random Number Generator
UDP	User Datagram Protocol
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
URI	Uniform Resource Identifier
ZLL	Zigbee Light Link

C License of the Documentation

References

[1] *GNU Free Documentation License*. Free Software Foundation, November 2002. 53

[2] Carsten Bormann. Concise Binary Object Representation (CBOR) Sequences. Internet-Draft draft-ietf-cbor-sequence-02, Internet Engineering Task Force, September 2019. 10

[3] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. Number 7228 in Request for Comments. RFC Editor, May 2014. Published: RFC 7228. 3, 33

[4] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. Number 7049 in Request for Comments. RFC Editor, October 2013. Published: RFC 7049. 9

[5] Minhaj Ahmad Khan and Khaled Salah. IoT security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395 – 411, 2018. 4

[6] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDoS in the IoT: Mirai and Other Botnets. *Computer*, 50(7):80–84, 2017. 4

[7] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Dan Boneh, editors, *Advances in Cryptology - CRYPTO 2003*, volume 2729, pages 400–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 11

[8] Ryan Levick. Why Rust for safe systems programming, July 2019. 24

[9] Zhen Ling, Kaizheng Liu, Yiling Xu, Chao Gao, Yier Jin, Cliff Zou, Xinwen Fu, and Wei Zhao. *IoT Security: An End-to-End View and Case Study*. 2018. 5

[10] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Communications Surveys Tutorials*, 21(3):2702–2733, 2019. 3

[11] Mani Pareek and Buriya Sushil. A Study of Link Layer Protocols in IOT. *International Journal on Future Revolution in Computer Science & Communication Engineering*, 4(2):355–359, February 2018. 6

[12] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. *IoT Goes Nuclear: Creating a ZigBee Chain Reaction*. 2016. Published: Cryptology ePrint Archive, Report 2016/1047. 4

[13] Ishtiaq Rouf, Hossen Mustafa, Miao Xu, Wenyuan Xu, Rob Miller, and Marco Gruteser. Neighborhood watch: Security and privacy analysis of automatic meter reading systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 462–473, 2012. 5

[14] Robert Scammell. Mikko Hyppönen: Smart devices are "IT asbestos", October 2019. 1

[15] Jim Schaad. *CBOR Object Signing and Encryption (COSE)*. Number 8152 in Request for Comments. RFC Editor, July 2017. Published: RFC 8152. 10

[16] Göran Selander, John Mattsson, and Francesca Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-selander-ace-cose-ecdhe-13, Internet Engineering Task Force, March 2019. 10, 12, 14, 46, 56

[17] Göran Selander, John Mattsson, Francesca Palombini, and Ludwig Seitz. *Object Security for Constrained RESTful Environments (OSCORE)*. Number 8613 in Request for Comments. RFC Editor, July 2019. Published: RFC 8613. 15, 16, 18, 19, 30, 46, 56, 57

[18] Y. Seralathan, T. T. Oh, S. Jadhav, J. Myers, J. P. Jeong, Y. H. Kim, and J. N. Kim. IoT security vulnerability: A case study of a Web camera. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 172–177, February 2018. 5

[19] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. Number 7252 in Request for Comments. RFC Editor, June 2014. Published: RFC 7252. 8, 9, 31, 56

[20] Stack Overflow. Stack Overflow Developer Survey 2019, April 2019. 24

[21] The Rust Core Team. Announcing Rust 1.0, May 2015. 20

[22] The Rust Release Team. Announcing Rust 1.36.0, July 2019. 32

[23] Anna Triantafyllou, Panagiotis Sarigiannidis, and Thomas D. Lagkas. Network Protocols, Schemes, and Mechanisms for Internet of Things (IoT): Features, Open Challenges, and Trends. *Wireless Communications and Mobile Computing*, 2018:1–24, September 2018. 6

List of Figures

List of Tables

Listings