Authentication and Authorization for Constrained Environments

Master Thesis

Urs Gerber

August 2018

Thesis supervisors:

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

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

Acknowledgements

I want to thank my family and friends for their continuous support during the duration of this project. It is due to them that I was able to participate in this master class.
I also want to express my thanks to Arnaud Durand and Professor Pasquier-Rocha, my thesis supervisors, who have provided me with the opportunity to work on this project and always pointed me in the right direction during the course of this project.

Abstract

In this work, we present a proof-of-concept implementation for the protocols and entities encompassed by the *Authentication and Authorization for Constrained Environments* (ACE) framework proposed in a working document of the Internet Engineering Task Force. The ACE framework is designed as an extension to the OAuth 2.0 authorization framework, which defines protocols how clients access protected resources on remote servers on behalf of a resource owner. The ACE framework adapts the protocols and primitives for use in the Internet of Things (IoT) where networking nodes can be very limited in terms of processing power and memory capacity. We provide a Python implementation for all ACE entities, i.e. authorization server, resource server and client. In order to achieve confidential communication between the client and resource server, we implement an application layer security protocol based on the Diffie-Hellman key exchange. Additionally, we demonstrate a resource server implementation capable of running on a constrained device. We increase the level of security for the embedded resource server by using a secure crypto element to perform asymmetric cryptography computations.

Keywords: Authorization, Authentication, OAuth, Web of Things, IoT

Table of Contents

1 Introduction

1.1 Motivation

There are a vast multitude of online services which provide the possibility for end users to store and access resources such as images, documents or other personal information. In most cases, users will have to go through a process of registration, where they assign themselves a unique identifier and passphrase or password. The set of unique identifier and password is called the *credentials* of the user for that particular service.

Whenever an end user wants to access a protected resource on a service, it can do so in two ways. In the simplest case, the user accesses the service's resource through the website of the service by supplying its credentials. The service then verifies the credentials and grants access to the resource. In another scenario, the user accesses the resource through a separate client, such as an application running on a smartphone. This client may be a first-party client provided by the owners of web service, or it may be a third party client.

In order for a third-party client to access the protected resource, the user would have to supply its credentials for the web service to the client, which the client would then forward to the server hosting the protected resource. While this may not be a problem for first-party clients, where the user probably trusts the client to keep its credentials secret, that may not be the case for third-party clients. Sharing credentials with potentially malicious third-party clients can be problematic due to various reasons. For example, the client could get unrestricted access to all of the originator's protected resources. Furthermore, the client could, willingly or unwillingly, disclose the credentials to unintended parties.

To mitigate these disadvantages, authorization frameworks such as *OAuth 2.0* provide mechanisms and protocols where authorization is granted to clients in a controlled way and access to resources can be managed granularly.

OAuth and related frameworks orchestrate and define protocols on how third-party applications access external data on behalf of a resource owner. In simplified terms, the authorization to gain access to a resource is delegated to a token. The user authenticates with some service and in turn receives an access token. In the case of *OAuth* 2.0, however, it is crucial that this token is not compromised or stolen. If the token is leaked, any party in possession of the token can impersonate the party the token was originally issued for. This kind of token is often called a *Bearer* token, since any party who bears the token

can use it. The use of Bearer tokens makes the use of properly secured communication channels compulsory. Due to this, the OAuth 2.0 specification makes the use of HTTPs mandatory for all implementations.

While HTTPs fulfills the requirements to keep these Bearer tokens secret from potential adversaries by encrypting the request parameters along with the token itself, it relies on the existence of a huge *Public Key Infrastructure* (PKI). The PKI defines roles and protocols that web servers and clients such as a web browser use to setup secure communication channels. However, access to this PKI can not always be guaranteed, or may not even be wanted. An example of this are *Internet of Things* networks.

1.1.1 Internet of Things & Constrained Devices

The *Internet of Things* (IoT) is a very general term used to describe the network of everyday interconnected devices. Similar to how the Internet connects terminals such as personal desktop computers, notebooks, servers or smartphones, the Internet of Things is the network connecting everyday physical objects equipped with networking functionality such as home appliances, sensors or even wearables and as such builds the network layer for the *Web of Things*.

The Web of Things (WoT) encompasses protocols and mechanisms that describe how IoT devices expose and access data on other IoT devices. It builds upon the concepts and protocols used in the *World Wide Web* and thus, many protocols initially developed for the Internet can be reused. Given that many IoT devices use sensors whose value can be shared with other IoT devices, there is a need for a standardized method to access these values. However, while being similar to traditional devices connected to the Internet, IoT devices come with inherent constraints that have an impact on how authorization frameworks such as OAuth 2.0 operate.

Processing Power The processors found in IoT devices are usually very limited in terms of processing power. This has a big influence into what kind of security can be achieved while communicating with the device, since the cryptographic computations required to secure communication can be computationally intensive.

Memory Compared to traditional Internet devices, IoT devices can be very limited with respect to available memory. Messages exchanged as part of a traditional OAuth 2.0 protocol flow may be too large for the IoT device to process.

Connectivity It is not unusual for IoT devices to be deployed in the field and thus draw their power from a battery. In order to achieve a long operating time, the messages exchanged in the authorization protocol should be kept as concise as possible. Transmitting long messages results in the transmission hardware to draw more energy from the battery, especially for wireless connectivity.

Despite being networking capable, nodes in IoT networks may not be connected to the Internet and in turn may not have access to the public key infrastructure which is required to securely transport messages between nodes in OAuth 2.0.

User Interface In traditional OAuth 2.0, whenever there is a need for a resource owner to authorize a client to access a resource, the user is presented with a rich user interface guiding the user through the approval process. However, many IoT devices are not equipped with the means to display such user interface.

All these challenges for authorization in IoT lead to a requirement of a new authorization and authentication framework that is built with constrained devices in mind.

In [1], Seitz et. al. propose an authorization and authentication framework based on the protocols and entities of OAuth 2.0, called *Authentication and Authorization for Constrained Environments* or ACE. The proposed ACE framework takes into account the limited resources available to constrained devices by altering some of the building blocks used in OAuth 2.0. For example, it replaces payload type from the text based *JSON* format to the binary *CBOR* (Concise Binary Object Representation) format, which reduces message size and in turn requires less power to transmit.

1.2 Goal

With this work, we want to contribute a proof-of-concept implementation of the protocols covered by the ACE framework written in the Python programming language. While there exists an early work in progress Java implementation of the ACE framework provided by the authors of [1], we find value in an easy to use Python implementation, given the strong presence of OAuth solutions written in the language.

Along with the Python implementation, we present an implementation for constrained devices written in C. These devices may not be capable of running a full Linux software stack and in turn may not be able to execute Python or Java code. As of the time of this writing, we are not aware of any embedded implementation capable of running on a constrained device.

It is important to note that the ACE framework still only exits as an Internet Engineering Task Force (IETF) working documentation and as such is subject to changes all the time. As a consequence, our implementation is based on a series of related RFCs and IETF drafts and is therefore expected to change as the framework matures.

The ACE framework encompasses a multitude of networking protocols and security mechanisms. Throughout this report, we will make references to networking protocols and cryptography related algorithms and mechanisms that are required to build an implementation of the entities proposed in the ACE framework. In the following chapter, we give a brief introduction into the networking protocols and cryptographic primitives relevant to the understanding of this report.

2
Theoretical Background

In this chapter, we will provide a short overview of basic concepts which we deem helpful to the understanding of the mechanisms and protocols described in this report.

2.1 Network Protocols

Almost all of todays Internet traffic is based on protocols defined in the *Internet Protocol* suite, which is often also referred to as *TCP/IP*. This suite of protocols describes how data is transferred from the origin to the intended destination. The protocol stack is arranged in a layered architecture, as illustrated in Figure 2.1.

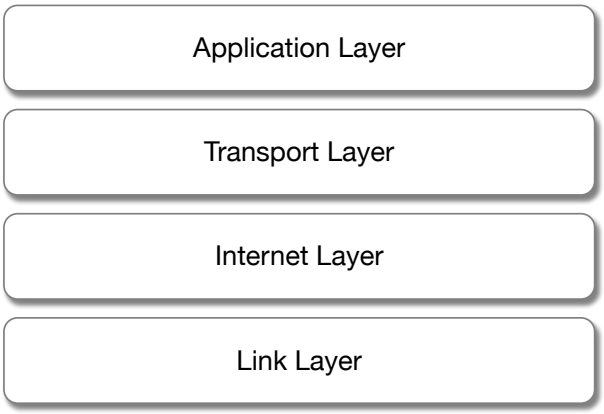

Fig. 2.1: Layers of the Internet Protocol suite

The layered architecture allows for more modular implementations, where upper layers can use services provided by the lower layers while introducing new functionality for the layers above. Both the Link layer and Internet layer are of lower importance for the understanding of this report. Consequently, we will focus on protocols used in the *Transport* and *Application* layer.

Transport Layer The Transport Layer provides messaging services for the overlying Application Layer. As such, protocols in this layer are responsible for transporting messages from one party to another while fulfilling certain requirements. The *Transmission*

Control Protocol (TCP) is a connection-oriented protocol which provides services for a reliable data transport. Thus, TCP packets are guaranteed to arrive in the same order that they were sent in. Additionally, TCP incorporates a retransmission mechanism to make sure that packets arrive at their intended destination even if a packet is lost during transit. In contrast, the *User Datagram Protocol* (UDP) is a connection-less protocol and does not provide reliable transport of messages. As a result, the overhead generated by UDP transmissions is much smaller compared to TCP. The difference in service provided by these protocols results in a diversified use pattern, where application protocols that require reliable data transport use TCP and applications that require a fast and concise message exchange use UDP.

Application Layer Application Layer protocols rely on the services provided by the underlying Transport Layer and expand on the mechanisms of the lower layer by introducing their own semantics and structure required to fulfill the requirements of the implemented application.

A prominent example of an Application Layer protocol is the *Hypertext Transfer Protocol* (HTTP). As specified in [12], HTTP is a text-based protocol which uses TCP to reliably transmit messages. HTTP is used by most web servers on the Internet to deliver content to a user's browser. The text-based nature of HTTP allows the traffic generated by the protocol to be analyzed very easily. As a consequence, all data related to a HTTP message, for example a message generated from a web server delivering the content of a website to a user, can be read by anyone on the same transmission medium. Network traffic analyzers can easily retrieve the address of the website that is being accessed as well as sensitive content provided by the user in the case of a file upload or login procedure. For this reason, the HTTP protocol was extended to use *Transport Layer Security* (TLS), specified in [3], which encrypts most of the message fields in an HTTP message. In addition, this extended version of HTTP called HTTP Secure (HTTPs) provides means for the communicating parties to mutually authenticate each other, such that both parties can make sure that they are communicating with the intended party and not an attacker on the shared medium.

An inherent disadvantage of HTTP is its text-based format. The fact that parameters in the protocol are formatted and stored as textual strings results in a large message footprint and long transmission times. In environments where communication nodes are very constrained in terms of processing power and memory, the messages generated by the HTTP protocol may even be unprocessable due to their inherent size.

To combat this circumstance, the *Constrained Application Protocol* (CoAP) was introduced and specified in [13]. In contrast to HTTP, CoAP uses a binary message format instead of textual strings to format the message parameters. This results in smaller messages being transferred between communicating parties and complies with the inherent lack of processing power and memory capacity of constrained devices. Additionally, CoAP messages are sent with UDP as the underlying transport protocol and thus omits the overhead associated with TCP. Due to the lack of a reliable transport protocol, CoAP introduces mechanisms on the Application Layer to achieve reliability requirements.

2.2 Authentication & Authorization

In the context of network communications, *authentication* describes the process of verifying the identity of a communicating party. To illustrate, we use the process of a user logging in on a web service. For this scenario, we assume that the user previously registered to the web service using a public unique identifier, usually an email address or user name, and a secret passphrase or password. In order to log in on the web service again, the user authenticates himself by providing that same unique identifier and password again. Assuming that said user is the only one in possession of the secret password, the website considers the user authenticated.

In contrast to authentication, authorization is solely concerned with what a communicating party is allowed to do. There may be multiple scopes or levels of access associated with a protected resource designed to limit the access of a party to that resource. Both authentication and authorization are important concepts for the explanations and expositions in the later parts of this report.

2.3 Cryptography

As mentioned previously, secure communication between parties is of great importance in today's Internet traffic. With increasing awareness of the dangers of unsecured message exchange, there is a surge in adoption of cryptographically secured communication. Making sure that no unintended party can read or alter a message in transit usually involves multiple cryptographic concepts. In this section, we will discuss the cryptographic primitives relevant to our implementation.

As indicated by Paar and Pelzl in [22, p. 263], whenever a party A wants to send a message x to party B over an unsecured channel, there are usually three requirements that A and B want to be fulfilled. Firstly, A may not want the message x to be read by other parties on the same open channel. This requirement is called *confidentiality*. In network communication, confidentiality is usually achieved using *encryption*, the basics of which we provide in the following paragraph. The second requirement is message *integrity*. B wants to make sure that message x has not been altered while in transit over the open channel. Lastly, party B might want to verify that message x was indeed sent by party A and not anybody else on the open channel, a requirement called *authentication*.

2.3.1 Encryption

The goal of obscuring the message x so that it can only be read by the intended parties can be obtained by *encryption*. Encryption is the process of transforming an original *plaintext* message into an obscured *ciphertext* which is designed to be unreadable for parties not involved in the bilateral encryption process. The general scheme for encrypting a plaintext message is illustrated in Figure 2.2.

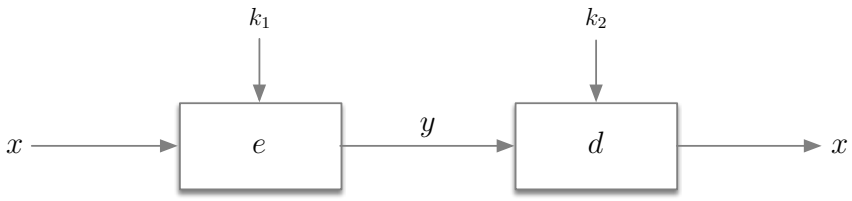

Fig. 2.2: Encryption scheme for message x. Figure adapted from [22, p. 150].

A encrypts its plaintext message x to obtain the encrypted ciphertext y using an encryption function e by computing $y = e_{k_1}(x)$, where k_1 is the encryption key. The now obscured message y can now be transferred over an insecure channel to the receiving party B. To decrypt the ciphertext message y, B must apply an appropriate decryption function d and a correct decryption key k_2. Formally, B obtains the original plaintext x by computing $x = d_{k_2}(y)$.

While there are a vast multitude of encryption and decryption function pairs e and d, all of these functions are either part of *symmetric encryption* or *asymmetric encryption*. Both of these categories imply some constraints on both the encryption and decryption functions e and d, as well as on the encryption and decryption keys k_1 and k_2.

Symmetric Encryption In symmetric encryption schemes, the encryption key and decryption key are the same, $k_1 = k_2 = k$. This common key k is often also called a *secret key*, or *shared secret*. That means, for a symmetric encryption scheme to work, both the sending party A, and the receiving party B must be in possession of a same shared key k. As a consequence, both A and B must somehow come to agreement about what key k they want to use in their secured communication. This can be achieved by exchanging this shared secret over a secured channel, or alternatively, the shared secret could be pre-established out of bounds.

One of the most commonly used symmetric encryption algorithms today is the *Advanced Encryption Standard* (AES), based on a proposal by Daemen and Rijmen in [11]. AES is a block cipher with a fixed block size of 128 bits, or 16 bytes and works with key sizes of 128, 192 and 256 bits, or 16, 24, and 32 bytes. While the inner workings of AES are indeed very interesting, we omit the implementation details of AES at this point as they are out of the scope of this report. We provide more information about AES in Section 5.2.1.

As mentioned previously, all symmetric encryption schemes suffer from the fact that both parties must be in possession of the same shared secret in order for the communication to be secure. *Asymmetric encryption* mitigates this disadvantage.

Asymmetric Encryption Asymmetric encryption schemes, often also called public-key encryption schemes, differ from symmetric encryption in the fact that the encryption key k_1 and decryption key k_2 are not the same. Instead, public key cryptography is built on the principle that only the decryption key of the receiving party must be kept secret, while the encryption key can be readily disclosed to any party interested in secure communication with the receiving party.

An example of an asymmetric encryption scheme is illustrated in Figure 2.3.

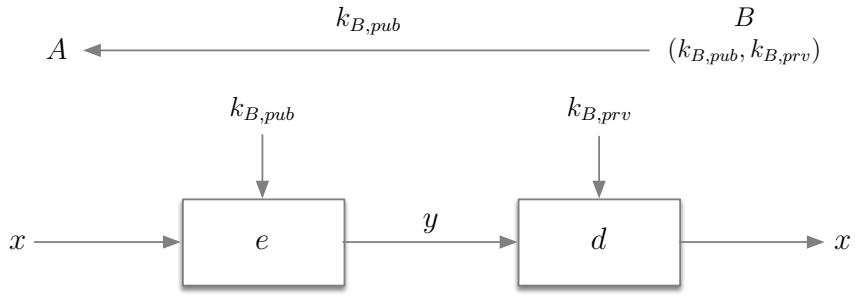

Fig. 2.3: Asymmetric encryption scheme. Figure adapted from [22, p. 152].

Before secure communication starts, the receiving party B creates an associated key pair consisting of a *public* and *private* key, $(k_{B,pub}, k_{B,prv})$. These keys are mathematically constructed in a way such that any message encrypted with the public key $k_{B,pub}$ can only be decrypted by B using its associated private key $k_{B,prv}$. At the start of the protocol, party B discloses its public key $k_{B,pub}$ to party A. In contrast to symmetric encryption, the exchange of the public key is not required to happen over a secure channel, since messages can only be encrypted using the public key.

Once A has received B's public key, party A encrypts the message x using the public key of party B by computing $y = e_{k_{B,pub}}(x)$ to obtain the ciphertext y. The resulting ciphertext is sent over the unsecured channel to party B, who then decrypts the ciphertext y using its private key, formally $x = d_{k_{B,prv}}(y)$.

As long as B's private key is kept secret, only B will be able to decrypt messages encrypted with the associated public key.

Man in the Middle Attacks While this protocol very effectively negates the need for a shared secret, it is vulnerable to *Man in the Middle* attacks. This kind of attack occurs when a malicious party M intercepts the transfer of B's public key to A over the open channel [22, p. 342]. Instead of simply forwarding B's public key to A, the malicious party M itself generates a public-private key pair and transmits its own public key to A. Without any additional countermeasures, A receives the public key of M but believes it to be the public key of party B. A now continues to encrypt the message x intended to only be read by B and transmits it over the open channel. M again intercepts this message and can decrypt it using its own private key. To complete the message exchange, M then encrypts the message using the public key of B, which it intercepted earlier, and forwards the message to B. If M's attack is successful, neither A nor B can know that their bilateral communication was intercepted and decrypted by a third party. The problem lies in the fact that A was not able to verify that the received public key was indeed the one which belongs to B.

One solution to this problem is the introduction of a public key infrastructure (PKI). The PKI is set up as centralized trust model where entities called *Certificate Authorities* (CAs) issue digital *certificates* for the binding of a communicating party to its public key [22, p. 344]. In simple terms, the certificate proves that a public key belongs to a certain party. This certificate, which contains the public key and identification information about

the party it belongs to, can then be transferred in the place of the public key. As long as A trusts the issuer of the certificate, it can validate the certificate and thus be confident that it is indeed communicating with the intended party B. Digital certificates build upon the foundations of digital signatures, which we explain in the following paragraph.

2.3.2 Digital Signatures

While encryption provides message confidentiality, *digital signatures* provide a mechanism to protect the *integrity* and prove the *authenticity* of a message sent over a communication medium. Using digital signatures it is possible for the recipient of a message to determine whether the message has been altered in transit and whether the claimed sender is the original author of the message [22, p. 260].

In order to achieve these goals, digital signatures employ an asymmetric encryption scheme. We illustrate the general principle of digital signatures in Figure 2.4.

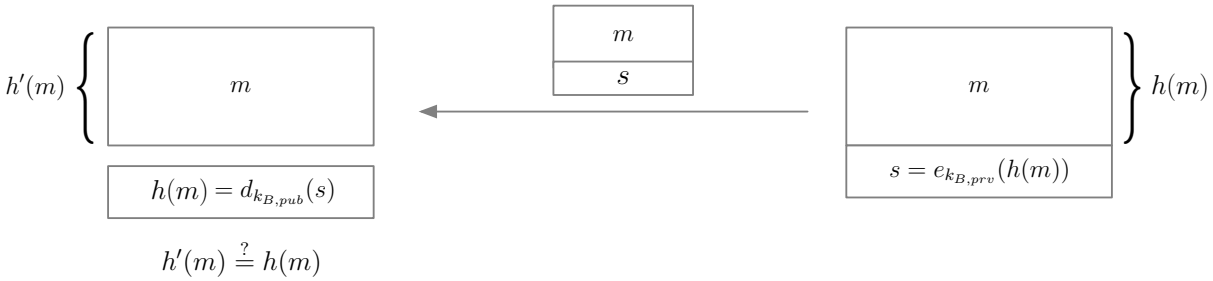

Fig. 2.4: Verifying the authenticity and integrity of a message using digital signatures.

We assume that party B wants to send a message m to party A. The content of message m is not sensitive so it is possible to transmit it openly over the unsecured channel. However, party A still wants to make sure that m has not been altered in transit and was indeed sent by party B. To this end, party B does not simply transmit m, but first computes the digital signature s of message m. It generates a key pair $(k_{B,pub}, k_{B,prv})$ and discloses its public key $k_{B,pub}$ to A prior to the message exchange. In this case, we presume the absence of a man in the middle attack, meaning that A trusts that $k_{B,pub}$ is the genuine public key of B.

The digital signature s of the message m is computed by first calculating a hash $h(m)$ of message m. While m can be arbitrarily long, the resulting hash $h(m)$ has a fixed length depending on the hash function h and is usually only a few bytes long. The intent of hashing the message first is to reduce the input to the asymmetric encryption algorithm in the following step.

After the hash is computed, B calculates the signature s of message m by encrypting the hash using its private key, or formally $s = e_{k_{B,prv}}(h(m))$. We should note that this is in direct contrast to the asymmetric encryption protocol described in previous paragraphs, where the party communicating with B encrypts its payload with the *public* key of B.

The computed signature s is then appended to the original message m and transmitted over the unsecured channel. On the other end, A receives the tuple (m, s) and can now

verify the integrity and authenticity of m using the digital signature s. In a first step, A also computes a hash $h'(m)$ from the message m by applying the exact same hash function h that was used by B when the signature was computed. In a second step, A decrypts the signature s using B's public key to obtain the original hash $h(m) = d_{k_{B,pub}}(s)$. To verify that message m was not altered since the time it was created by B, A simply has to compare $h(m)$ and $h'(m)$. If both hashes are, the same, A can be sure that m was not altered in transit and B was the originator of the message.

2.3.3 Key Exchange

As mentioned previously, asymmetric cryptography tends involve more computationally intensive operations compared to symmetric cryptography [22, p. 156]. The computational power required for encrypting a message increases with the length of the message with the consequence that long messages require more computational effort to be encrypted than short messages.

Additionally, keys in asymmetric encryption schemes are required to be much larger in size than in their symmetric counter parts in order to provide the same order of security [22, p. 156]. In the context of constrained devices, where computation power is very limited, and smaller keys lead to shorter messages when keys need to be transferred, symmetric encryption is often preferred over asymmetric encryption.

Instead of relying on asymmetric cryptography for the encryption process itself, there is an approach where symmetric keys can be securely established between communication parties using an asymmetric key exchange protocol. In a key exchange protocol, the goal is to establish a *shared secret* between communication parties over an unsecured channel. The shared secret is a symmetric key which is used by both parties participating in the encryption scheme to encrypt and decrypt messages using a symmetric encryption algorithm.

To compute this shared secret, the two parties both generate an asymmetric key pair consisting of a public and private key. They then exchange their respective generated public keys to the other party. After having received the public key, both parties compute a shared secret by combining their respective private key of the generated key pair with the received public key of the other party. The computation required for this operation is dependent of the desired cryptosystem. One example of a key exchange protocol is the Diffie-Hellman key exchange, a version of which we detail in Section 5.3.1. As in any asymmetric cryptography protocol, the public keys exchanged during the protocol need to be authenticated in order to mitigate man-in-the-middle attacks.

3 Authorization using OAuth 2.0

The ACE framework relies heavily on the primitives and protocols introduced in the OAuth 2.0 authorization framework proposed by Hardt et. al in [2]. In this chapter, we illustrate the concept behind OAuth 2.0 and depict some of the protocols that it proposes.

3.1 Overview

The OAuth 2.0 framework proposes a standardized method for clients to access resources on remote servers. OAuth distinguishes four roles that participate in the authorization and resource access process: *resource owner*, *client*, *authorization server* and *resource server*.

Resource Owner (RO)
: The resource owner is originator of the protected resource being accessed. The resource owner is the sole entity with the authority to authorize clients to access a protected resource.

Client (C)
: The client is the application trying to access a protected resource on behalf of the resource owner. A client can only perform successful requests to the protected resource once the client has been authorized by the resource owner.

Resource Server (RS)
: The resource server is the host where the protected resource resides. Whenever a request to a resource is made, the resource server needs to verify that the requesting client is authorized to access the resource.

Authorization Server (AS)
: The authorization server is the orchestrating entity in the OAuth framework. As such, it authorizes clients on behalf of the resource owner by issuing access tokens, which represent a time limited access grant to a protected resource for one particular client.

How these roles interact with each other is depicted in Figure 3.1.

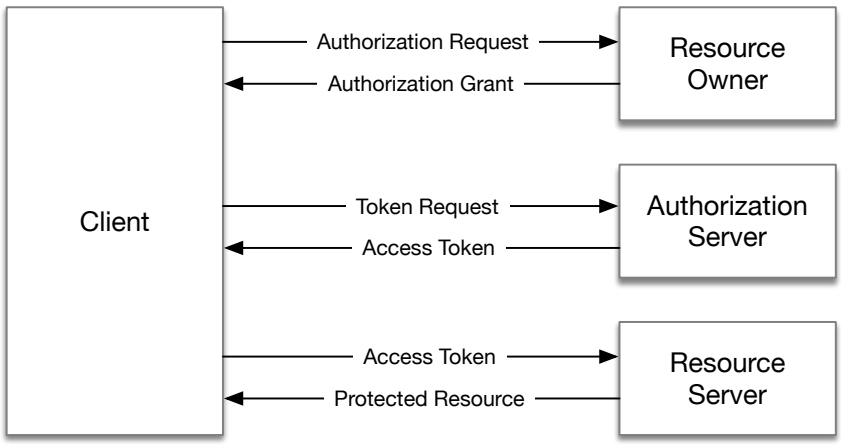

Fig. 3.1: OAuth 2.0 General Protocol Flow

The yet unauthorized client initiates the protocol by performing an *authorization request* to the resource owner. In order for the resource owner to make an informed decision, this request includes information about which resources the client whishes to access. At this point, the resource owner has the opportunity to either authorize or deny access to the resources requested by the client. If the resource owner authorizes the client, the client will receive an authorization grant which is a representation of authorization on behalf of the resource owner.

Once the client has received the authorization grant, it requests an access token from the authorization server in exchange for the previously obtained authorization grant. The authorization server will authenticate the client and verify the authorization grant. To authenticate the client, it must be pre-registered with the authorization server to setup *client credentials*. The client credentials consist of a unique public *client identifier* and a *client secret*. If the authorization server has verified that this is a registered client and the authorization grant is valid, it will return an *access token*. This access token represents a time limited authorization to access a particular protected resource. The client then stores the access token for future reference.

To actually access the protected resource on the resource server, the client performs a request to the endpoint of the resource and includes the access token it received from the authorization server in the previous step. After checking the validity of the access token, the resource server will respond with either the protected resource or it will deny access to the client in case the access token is no longer valid.

3.2 Access Token

The access token obtained by the client from the authorization server is of central importance in OAuth 2.0. It serves as means of authorization to access a protected resource without the need for the resource owner to present its credentials to the client. The access token can be of any form, provided that the resource server is able to confirm its validity and grant access to the requested protected resource. As such, the token is called *opaque* to the client. The client should not be concerned with what is contained in the access

token, as it only needs to store and present it to the resource server whenever it needs to access a protected resource. The access token may be self-contained, meaning that it carries all the authorization information necessary for the resource server to accept it, or it may only be a referential string that the resource server then uses to look up the authorization associated with that reference either locally or with the help of the authorization server.

While the OAuth 2.0 specification does not make requirements or suggestions concerning the form of the access token, many OAuth 2.0 implementations use access tokens adhering to the *JSON Web Token* (JWT) standard specified in [20]. In this standard, the access token is a transparent digitally signed JSON object string which contains the relevant authorization information for the access token, such as requested scopes or expiry date and time. We will further explain the structure of the access token in our implementation in Section 4.4.2. In the OAuth 2.0 protocol flow, the access token is sent along with every request from the client to a protected resource in the form of an HTTP header.

Security Considerations It is important to note that the access token should always be kept secret by the client. The access tokens issued by OAuth 2.0 authorization servers are usually *Bearer* tokens, meaning that any entity who presents the token is allowed to access the protected resources covered in the access token.

This implies that at least the access token request from the client to the authorization server, as well as the request to the protected resource bearing the access token must be performed over a secure channel. This is the reason why the OAuth 2.0 specification requires implementations of these endpoints to be secured by HTTPs in which case the access token is encrypted along with other request headers.

3.3 Grant Types

The OAuth 2.0 specification proposes multiple types of protocol *flows*, often also referred to as *grant types*, designed to cater to the specific capabilities of the client. The general protocol flow discussed in Section 3.1 is a very abstract flow that needs to be adapted to the capabilities of the client.

OAuth distinguishes two types of clients, *confidential* and *public* clients. Confidential clients are clients which can keep their client credentials secret, such as applications running on a remote web server. In contrast to confidential clients, public clients, such as an application running on a device or in the browser of the resource owner, can not be trusted to always keep their client credentials secret, since they could be inspected for their credentials at runtime.

For this report, we restrict our discussions to confidential clients and examine two particular protocol flows proposed in the OAuth 2.0 specification.

3.3.1 Authorization Code Flow

The *authorization code flow* is targeted at clients running on a remote web server. It presumes the presence of a user agent on the resource owner's side, such as a web browser, which can be redirected to an address specified by the authorization server. We illustrate the authorization code flow in Figure 3.2.

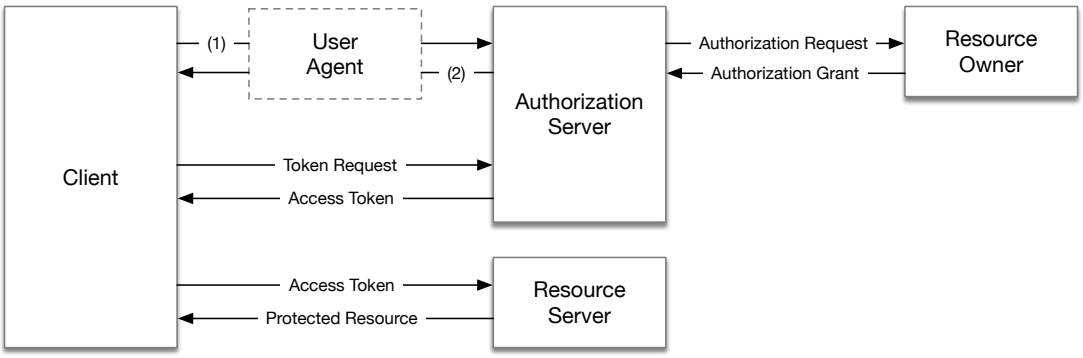

Fig. 3.2: OAuth 2.0 Authorization Code Flow

In this code flow, the resource owner grants authorization to the client via a mediating authorization server. We assume that the client has been registered with the authorization server in a previous step, meaning that the authorization server has assigned a unique client identifier and client secret to the client.

The client navigates the resource owner's user agent to the *authorization* endpoint of the authorization server offered by the service that the client whishes to access. The client has to provide its public identifier and the intended scopes it wants access to. The authorization server then prompts the resource owner to authenticate himself by providing the credentials of the resource owner. It is important to note that the resource owner only provides its credentials to the authorization server of the service, not the client. The client never gets a hold on the resource owner's credentials.

Once the authorization server has authenticated the resource owner, it provides the resource owner with the possibility to review the resources the client whishes to access, and prompts the user to either authorize or decline the client. The authorization server then redirects the resource owner's user agent to a callback URL specified by the client in the initial authorization request, along with the result of the authorization and, in the case of approved authorization, an *authorization code*.

We should note that the initial authorization code request and subsequent redirects are all visible on the resource owner's browser, where all query and response parameters can be easily inspected. For example, the authorization server returns the authorization code as an URL query parameter in the redirect URL, which can be seen by the resource owner in the address bar of the browser.

Prior to accessing the protected resource, the client requests an access token from the authorization server by accessing the *token* endpoint on the authorization server along with providing the authorization code obtained in the previous step. The client will also have to supply its client credentials in order to authenticate itself to the authorization

server. The authorization server verifies the authorization code and client credentials and will generate a time limited access token for the client scoped to the resource it is trying to access.

Once the client has received the access token, it can query the resource server on the endpoint of the protected resource. Given that the client supplied a valid access token, the resource server will respond with the protected resource.

In contrast to the initial authorization code request, the access token request and subsequent resource request do not require a user agent. As a consequence, none of the information passed between the entities, such as the issued access token, are visible to the resource owner or the user agent. This increases the security of the protocol as potentially compromised browsers are not able to steal the access token.

3.3.2 Client Credentials Flow

As stated in the previous section, the authorization code flow is aimed for clients running on remote servers. It also presumes a resource owner that needs to authorize clients to access a protected resource on its behalf. In IoT networks, however, intermediate nodes often need to access resources on other nodes without the interactions of a resource owner. In these cases, where machine-to-machine communication is predominant, the client and the resource owner coincide. As a consequence, there is no need for a separate request for an authorization code. This adapted code flow is called the *client credentials flow*, which we have depicted in Figure 3.3

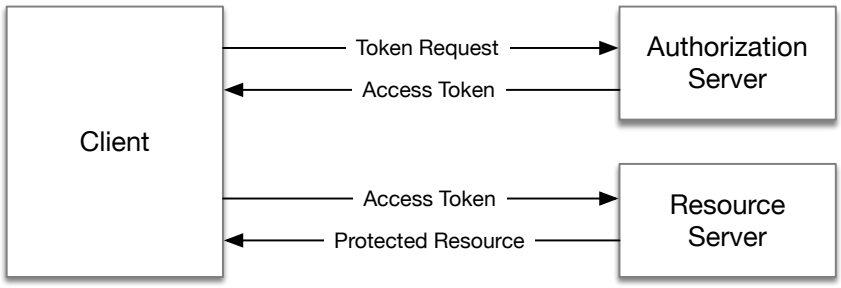

Fig. 3.3: OAuth 2.0 Client Credentials Flow

In the client credentials flow, a client is only able to access protected resources it owns. Alternatively, it can also access other protected resources given that there is a prearranged authorization between the client and authorization server. This makes it necessary for the client to be registered with the authorization server along with all the resources and scopes the client is allowed to access.

The client initiates the client credentials flow by posting a request to the *token* endpoint of the authorization server. In this request, the client needs to provide its eponymous client credentials in order for the authorization server to authenticate the client. The client also provides the scopes it wishes to access. Once the authorization server has authenticated the client and verified the requested scopes, it issues an access token.

Consequently, the client presents the previously obtained access token to the resource server which hosts the protected resource. The resource server then verifies the validity of the access token and responds with the protected resource.

The client credentials flow discussed above builds the foundation of the authorization protocol flow used in the ACE framework.

4
ACE Framework

As stated in the previous chapter, the ACE framework builds upon the fundamentals introduced by the OAuth 2.0 framework. In this chapter, we will introduce the changes the ACE framework proposes to the OAuth 2.0 framework in order for it to be usable on IoT networks with constrained devices.

4.1 Overview

The ACE framework as proposed by Seitz et. al. in [1] extends the OAuth 2.0 framework with the goal to expand the authorization flow to constrained devices in IoT networks. These types of devices are inherently limited in their processing capability or have to work in a very constrained memory environment. Furthermore, power consumption is of primary concern for devices deployed in the field relying on a battery for power delivery. This makes it necessary for the message exchange to be as concise and compact as possible. ACE addresses this by requiring the messages to be formatted as defined by the *Concise Binary Object Representation* or CBOR, which is a replacement for the text based *Javascript Object Notation* (JSON) in OAuth 2.0. We give a brief overview of CBOR in Section 4.4.

The ACE framework introduces the concept of *security profiles* to comply with the fact that devices in IoT networks need more flexibilty with respect to the message exchange mechanisms used in the authorization flows. While OAuth 2.0 solely relies on TLS (via HTTPs) to secure the communication between OAuth entities, these security profiles allow communicating nodes to specify their respective security and networking capabilities. The communicating parties will then agree on a security profile that best fits their capabilities. We provide insight into the implemented security profile in this project in Chapter 5.

Compared to bearer access tokens in OAuth 2.0, where any entity bearing the token can request the protected resources covered by that token, the access tokens issued by an ACE authorization server require the presenting entity to put forth *proof of possession* for that particular token. This mechanism makes it impossible for clients to use an access token issued for another client. We explain the structure and properties of an ACE access token in Section 4.5.3.1.

4.1.1 Protocol Flow

As mentioned in Section 3.3.2, the ACE protocol flow is based on the *client credentials* flow as proposed in the OAuth 2.0 specification. Clients in IoT networks often do not require an authorization grant from a resource owner to access a protected resource, since the client itself may be the resource owner or the resource server where the resources are hosted are in direct control of that specific client. The general ACE protocol flow is depicted in Figure 4.1.

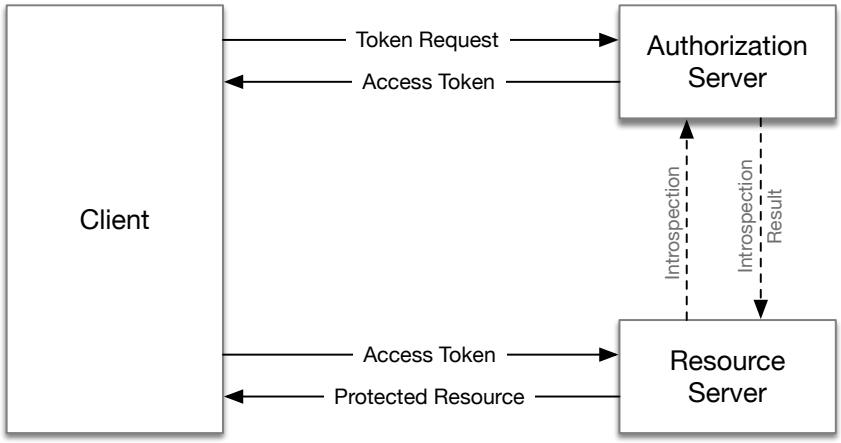

Fig. 4.1: ACE general protocol flow

In this scenario, we assume that the client already knows the address of the resource server which hosts the resource the client wants to access. The client requests an access token from the authorization server by posting its client credentials to the authorization endpoint at the authorization server. The authorization server will then authenticate the client using the presented credentials and return an access token scoped to the scopes and resources requested by the client.

After receiving the token, the client can use the token to access the protected resource on the resource server. The client makes a request to the endpoint of the protected resource and includes the access token it received from the authorization server in the previous step. The resource server then validates the access token and responds with the protected resource.

If the access token is not self-contained and thus is just a referential string, the resource server will *introspect* the access token with the help of the authorization server.

In the following sections, we present more details about the steps depicted in Figure 4.1.

4.2 Provisioning

The ACE framework requires both the initial token request (and response) as well as the access to the protected resource to be happening on a secured communication channel. As opposed to OAuth 2.0, nodes in IoT networks may not be involved in a common key infrastructure. As a consequence, there is a need for extensive provisioning of credentials

for the ACE entities (client, resource server, authorization server) prior to the start of the protocol flow.

Public Keys All participating ACE entities (client, authorization server and resource server) need to be configured with a public and private key pair which will be used to authenticate messages sent by these respective entities. The client and resource server will need to know the public key of the authorization server, since both the client and the resource server are required to verify the authenticity of the access token issued by the authorization server. We have illustrated the provisioned keys in Figure 4.2.

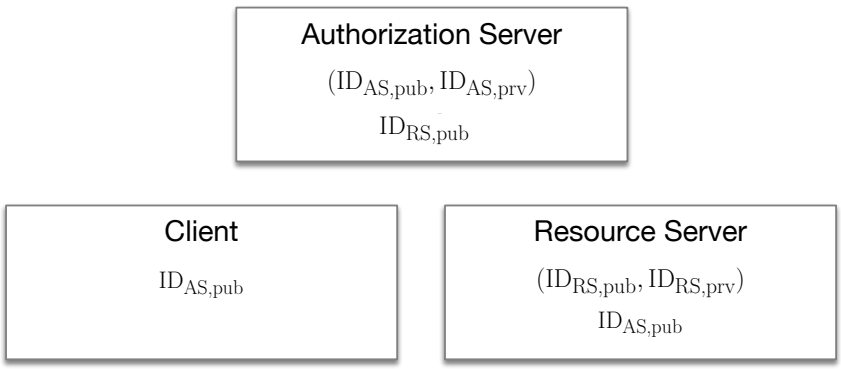

Fig. 4.2: Provisioned keys for the ACE entities

It is important to note that the client and resource server do not need to pre-establish each other's public keys for authentication. Given that there may be a vast number of resource servers in an IoT network, it would be unfeasible to pre-establish trust between every client and resource server in the network. Mutual authentication between resource server and client will be performed as part of the establishment of a secure context at the time of the access to the protected resource.

Credentials and Profiles Furthermore, the authorization server must know the credentials (client identifier and client secret) of the clients along with the resources and scopes on these resources the client is allowed to access. The authorization server must also know the security profiles supported by the resource server and client, as the authorization server will decide which security profile will be used in the communication between the resource server and the client.

4.3 Transport and Application Layer Protocols

The OAuth specification requires implementations of the specification to solely target HTTPs using TLS. Since HTTP 1.0 is a text based protocol, the ACE framework recommends the use of binary application layer protocols such as CoAP which employ smaller message sizes compared to textual protocols. CoAP usually runs on top of UDP. As a replacement for TLS, the ACE specification recommends to use DTLS to secure commu-

nication using CoAP messages. DTLS is an adaption of TLS designed to use UDP as its transport protocol instead of TCP.

However, in our implementation, we chose to use HTTP as the application layer protocol since the networking libraries implementing CoAP servers and clients showed poor support for DTLS secured communication at the time of writing this report.

4.4 Concise Binary Object Representation (CBOR)

The ACE framework requires implementations to change the format of messages from the text based `Javascript Object Notation` (JSON) to the `Concise Binary Object Representation` proposed by Bormann and Hoffmann in [8]. The benefit of replacing JSON with CBOR manifests itself in the significantly reduced message size and consequently reduced transmission power. Similar to JSON, CBOR specifies how primitives such as integers, text strings or key-value maps are encoded as binary objects such that the obtained binary object is as concise as possible.

For the purpose of this report, we refrain from explaining in detail how arbitrary objects are encoded as CBOR objects and instead introduce entities required by the ACE framework that use CBOR as the underlying encoding format. These entities include signature and encryption objects as well as an object that represents an access token. In Figure 4.3 we have illustrated the entities built on top of CBOR.

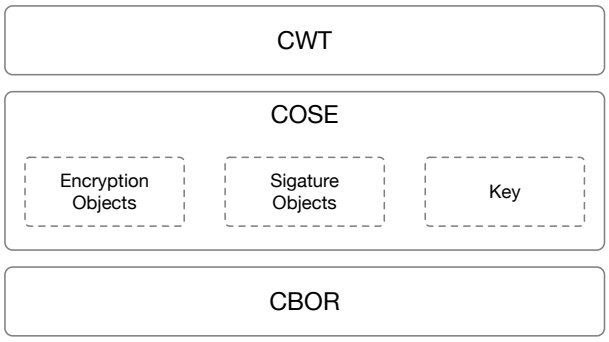

Fig. 4.3: The CBOR Object Signing and Encryption (COSE) standard is built on top of CBOR and can be used to encode a CBOR Web Token (CWT)

4.4.1 CBOR Object Signing and Encryption (COSE)

The CBOR Object Signing and Encryption (COSE) standard defined in [7] proposes standardized CBOR objects that represent cryptographic keys as well as objects that carry encrypted or digitally signed payload. In the context of the ACE framework, the primitives provided by COSE are used to transmit encrypted or signed messages between the ACE entities. In the following paragraphs, we introduce the format of objects relevant to our implementation of the ACE framework.

Key Object In certain messages being exchanged between ACE entities during the protocol flow, one entity sends a certain cryptographic key to another entity. The COSE standard proposes a method of encoding these keys such that both entities agree on the structure and semantics of the key object. The proposed structure for encoding an elliptic curve key is illustrated in Listing 4.1.

```
{
  / kty /  1: Key Type,
  / kid /  2: Key Identifier,
  / alg /  3: Algorithm Identifier,
  / crv / -1: Elliptic Curve Identifier,
  / x /   -2: x-Coordinate of public key,
  / y /   -3: y-Coordinate of public key,
  / d /   -4: Private key parameter
}
```

List. 4.1: COSE Key structure

The COSE key object is encoded as a CBOR map which is a simple array of key-value pairs. The keys of the map are short integer digits. It is easy to see how using integers as keys can result in smaller encoded messages compared to textual string keys in JSON. The `kty`, `kid` and `alg` parameters are general data about the encoded key specifying the type and identifier for the encoded key, respectively the identifier for the algorithm the key is being used with. The `crv`, `x`, `y` and `d` parameters are specific to elliptic curve keys and relate to the elliptic curve parameters of the encoded key.

Encryption Object Additional to encoding key objects, the COSE standard provides a binary encoding for an object carrying an encrypted payload. These objects are used in the ACE protocol flow whenever a message is required to be sent encrypted from one entity to the other. While COSE supports a wide variety of objects dedicated for encryption, our implementation only makes use of the `Encrypt0` structure. This structure can be used in cases where the recipient of the encrypted object is implicitly known and does not need to be included in the object. The COSE `Encrypt0` structure is a CBOR array containing three elements as illustrated in Listing 4.2.

```
[
  protected_header,   // CBOR Map encoded as byte string
  unprotected_header, // CBOR Map
  ciphertext          // Byte String
]
```

List. 4.2: COSE Encrypt0 Object

The `protected_header` parameter is a CBOR map encoded as a byte string which contains parameters about the ciphertext that need to be integrity protected by the encryption algorithm. As an example, an entry in this map could represent a key identifier for the key that was used to encrypt the ciphertext contained in the object, which the recipient can then use to retrieve the corresponding decryption key. In contrast, the `unprotected_header` parameter contains a CBOR map of values that are not required to be integrity protected but still related to the encrypted payload.

Signature Object In cases where encryption is not necessary, COSE defines a structure for objects that require a digital signature to be attached to the payload itself. The digital signature can be used by the recipient to verify that the protected content of the message has not been tampered with during transit. In our implementation, where we only have a single recipient for the digitally signed object, we use the COSE `Sign1` object whose structure is depicted in Listing 4.3.

```
[
  protected_header,   // CBOR Map encoded as byte string
  unprotected_header, // CBOR Map
  payload,            // Byte String
  signature           // Byte String
]
```

List. 4.3: COSE Sign1 Object

Similar to the COSE `Encrypt0` object, the `Sign1` object is a CBOR array with four instead of three elements. The header parameters use the same semantics as in the case of the COSE `Encrypt0` object. The `payload` parameter contains the integrity protected application data and the `signature` parameter contains the computed digital signature. The parameters covered by the digital signature include the `protected_header`, the `payload` and optional externally supplied data. In the context of an ACE protocol flow, this signature object can be used wherever data need to be integrity protected and authenticated, as is the case in the access token issued by the authorization server.

4.4.2 CBOR Web Token (CWT)

The ACE framework recommends that implementations use `CBOR Web Tokens` (CWTs) as the format for self-contained access tokens. A CWT (which has become an RFC during the course of this project) as defined in [9] is either an encrypted or digitally signed COSE object, leveraging the COSE encryption and signature structures. In our implementation, access tokens are represented by a digitally signed CWT since we only include public keys in the access token and thus encryption of symmetric keys is not required. A digitally signed CWT uses the COSE `Sign1` structure to form an access token as indicated in Listing 4.4.

```
[
  {
    / alg / 1: Identifier for the Digital Signature Algorithm
  }, // CBOR Map encoded as byte string, protected_header
  {
    / kid / 4: Identifier for key used to sign this CWT
  }, // CBOR Map, unprotected_header
  {
    / iss / 1: Issuer of access token (authorization server),
    ...
    / cti / 7: CWT Identifier,
  }, // CBOR Map encoded as byte string, payload
  signature // Byte String
]
```

List. 4.4: Structure of a CBOR Web Token

The `protected_header` map includes the `alg` parameter that identifies which digital signature algorithm was used to compute the digital signature included in the CWT. The `kid` parameter in the `unprotected_header` map contains an identifier for the key that was used to sign the CWT. Parties interested in verifying the authenticity of the CWT can use this key identifier to retrieve the correct public key of the issuer to verify the digital signature contained in the CWT. The `payload` parameter of the CWT contains the *claim set* of the access token which we explain in Section 4.5.3.1. The last element is the `signature` parameter containing the computed digital signature of the CWT. Recipients can use this digital signature to check the authenticity of the access token.

4.5 Protocol Flow

In this section, we will discuss the ACE protocol flow in detail by illustrating a case where a client wants to access a protected resource on a resource server after requesting an access token from the authorization server.

4.5.1 Token Request

At the start of the protocol, the client performs a POST request to the */token* endpoint of the authorization server. As required by the ACE framework, this request is sent over a secured channel and the client and authorization server need to mutually authenticate each other. In plain OAuth 2.0, this request is protected by TLS and the client would authenticate the authorization server using a TLS certificate. For constrained devices, the ACE framework requires the security profile to specify how the client and authorization server establish secure communication.

For the purposes of this report, and in our implementation, we assume that the communication between the client and authorization server is protected by a protocol similar to TLS. As a consequence, the client considers the authorization server authenticated, while the authorization server verifies the authenticity of the client with the aid of the client's credentials.

The request parameters sent to the authorization server by the client are depicted in Listing 4.5.

```
{
  "client_id": "ace_client_1",
  "client_secret: "ace_client_1_secret_123456",
  "grant_type": "client_credentials",
  "aud": "tempSensor0",
  "scope": "read_temperature,post_led",
  "cnf" : (K_PoP,pub formatted as COSE Key)
}
```

List. 4.5: Parameters sent in the token request

It is important to note that the parameters in Listing 4.5 are depicted in standard JSON format, rather than as an actual CBOR map, for notation and illustration purposes.

The `client_id` and `client_secret` parameters constitute the client credentials of the requesting client. This examples illustrates why this request must only be performed over a secured channel, since the client is sending its client secret, which should be kept confidential under all circumstances. If the client secret is disclosed, any party presenting the secret could impersonate the client, given that the client identifier is public.

The `grant_type` parameter instructs the authorization server to use a specific protocol flow, which in this case is the client credentials flow. The `aud`, short for *audience*, and `scope` parameters both refer to the protected resource the client wants to access. In this example, the client wants to access the `read_temperature` and `post_led` scopes on the resource server known to the authorization server as `tempSensor0`.

The `cnf` parameter refers to the *proof of possession* key the client wants the authorization server to bind to the access token. The structure of the key in the `cnf` parameter is explained in Section 4.4.1.

4.5.2 Proof of Possession

The ACE framework suggests for implementations of the protocol entities to use *proof-of-possession* (PoP) tokens instead of *Bearer* tokens as used in the OAuth 2.0 specification. While a Bearer token can be presented by any client that gets a hold of the token, proof-of-possession tokens can only be successfully presented to a resource server if the presenting client can demonstrate the possession of a certain cryptographic key.

To this end, the client instructs the authorization server to bind a key to the access token. This key is called the proof-of-possession key, or PoP key, which can either be symmetric or asymmetric. For the purpose of this report and our implementation, we omit the details for symmetric PoP keys. In the asymmetric case, the client itself creates a public-private key pair and includes the public part of the PoP key in the `cnf` parameter of the initial token request, as illustrated in Listing 4.5.

The authorization server will then include the supplied PoP key in the access token itself. In other words, the authorization server *binds* the PoP key to the access token.

Whenever the client wants to access a resource on a resource server, both parties are required to establish a security context to create a secure channel for their communication. The setup of this security context will only be successful if the client can prove to the resource server that the client in question knows the *private* key corresponding to the PoP key that was bound to the token by the authorization server. We discuss the establishment of a secure context in Chapter 5.

The proof-of-possession mechanism prevents other clients that get a hold of the access token from accessing resources associated with that token, since these clients do not know the private part of the PoP key bound to the access token.

4.5.3 Authorization Server

Once the authorization server has received the request as illustrated in Listing 4.5, it authenticates the client by inspecting the supplied client credentials. Furthermore, the

authorization server checks the `aud` and `scope` values that the client wants to access. Given that this particular client was provisioned to access the requested audience and scopes, the authorization server will issue an access token.

4.5.3.1 Access Token

The structure of an access token is depicted in Listing 4.6.

```
{
  "iss": "ace.as-server.com",
  "iat": 1533296406,
  "exp": 1533303606,
  "aud": "tempSensor0",
  "scope": "read_temperature,post_led",
  "cnf": (K_PoP,pub formatted as COSE Key)
}
```

List. 4.6: Claim set of access token

The key-value pairs in an access token are often referred to as *claims*. The `iss` claim denotes the domain of the authorization server which issued the token. The `iat` and `exp` claims state the time at which the token was issued and the time the token expires respectively. Both times are denoted in seconds since epoch time. The `aud` and `scope` claims refer to the same values that were requested by the client in the token request. The `cnf` claim, as explained in Section 4.5.1, represents the proof-of-possession key that the authorization server has bound to the access token, as instructed by the client.

The final structure of the access token is formed by including the claim set illustrated in Listing 4.6 as the payload of a CBOR Web Token as illustrated in Section 4.4.2.

4.5.4 Token Response

As explained in Section 4.4.2, when included in the actual response, the access token will be encoded as a CBOR map. Furthermore, the authorization server cryptographically signs the claim set in the access token using its private key $ID_{AS,prv}$. Since both the client and resource server have pre-established trust via the public key of the authorization server $ID_{AS,pub}$, both entities will be able to cryptographically verify that the access token was indeed issued by a trusted authorization server, and was not altered in transit. Since the access token does not contain sensitive information that should not be visible to other parties, it is not required for the access token to be encrypted. Consequently, the claim set will be wrapped in a `COSE Sign1` structure to form a *CBOR Web Token* or *CWT*, which includes both the claim set and the digital signature.

The access token, encoded as a CBOR web token, will then be placed in the token response as depicted in Listing 4.7.

```
{
  "access_token": "..." (omitted for brevity),
  "token_type": "pop",
  "profile": "coap_oscore_edhoc",
  "rs_cnf": (ID_RS,pub formatted as COSE Key)
```

```
6 }
```

List. 4.7: Token response from authorization server to the client

The `token_type` parameter in the response object indicates to the client that the access token included in the response is a proof-of-possession token, rather than a Bearer type token, as would be the case in OAuth 2.0. In the response object, the authorization server also includes the `profile` parameter, which instructs the client to use the specified security profile when accessing the protected resource on the resource server. In the example response in Listing 4.7, the security profile chosen by the authorization server is the `coap_oscore_edhoc` profile, which is an application layer end-to-end security profile. We discuss this security profile in Chapter 5.

The last parameter in the token response is the `rs_cnf` parameter. It includes the public key of the resource server hosting the protected resource, $\text{ID}_{\text{RS,pub}}$. The client will use this key to authenticate the resource server. It is necessary for the authorization server to include the resource server's public key since it is not feasible to pre-configure every client-resource server combination with each other's respective public keys.

It is important to note that the client is receiving the above mentioned response over a confidential and authenticated channel. As a consequence, the client can be certain that the received key $\text{ID}_{\text{RS,pub}}$ stated in the `rs_cnf` parameter is authentic. The authorization server then sends the response as depicted in Listing 4.7 back to the client, encoded as a CBOR map.

4.5.5 Client

Upon receiving the authorization server's response to the token request, the client inspects the response object and extracts the access token. As mentioned previously, the access token is a CBOR web token and as such bears the structure of a `COSE Sign1` object. Consequently, the client is able to verify the signature attached to the CWT using the public key of the authorization server – $\text{ID}_{\text{AS,pub}}$ – which was pre-installed on the client in a provisioning step. If the verification succeeds, the client can be certain that this particular access token was indeed issued and signed by an authorization server trusted by the client. It should be noted that while the client is able to inspect the claims in the access token, it is not able to change any of the claims in the access token, as doing so would be detectable by the resource server due to the access token's cryptographic signature.

Furthermore, the client also takes note of the security profile proposed by the authorization server. In our example, the authorization server instructs the client to use the `coap_oscore_edhoc` profile, for which the client requires the public key of the resource server associated with the protected resource covered by the access token. The `coap_oscore_edhoc` profile requires the client and resource server to establish a secure context which will then be used to create a secure channel between the communicating parties. To successfully establish such a context, the client is required authenticate the resource server before accessing the protected resource in order to confirm that the client is only accessing resource servers trusted by the authorization server. Omitting this

step could result in the client accessing tampered information from potentially malicious resource servers.

4.5.5.1 Token Upload

Before the client performs a request to the protected resource, it needs to upload the access token to the resource server. This step is required since the access token contains the proof-of-possession key that was bound to the access token, which will be used by the resource server to authenticate messages from the client to establish a secure context between the client and the resource server. We have illustrated the process of accessing a protected resource on the resource server in Figure 4.4.

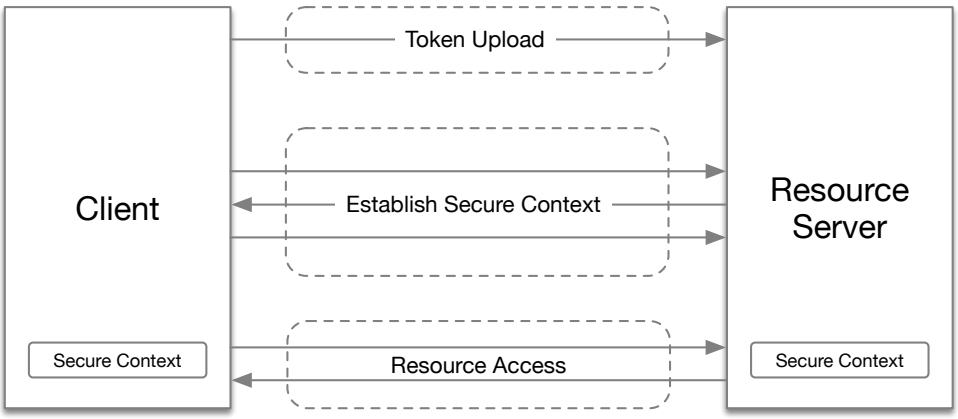

Fig. 4.4: Access to protected resource

To this end, the client sends the token to the `/authz-info` endpoint offered by the resource server, which is required to perform additional processing of the access token.

4.5.6 Resource Server

Once the resource server has received the access token from the client, it verifies the signature of the access token using the provisioned public key of the authorization server, $\mathrm{ID}_{\mathrm{AS,pub}}$. By doing so, the resource server is able to confirm that the access token was issued by a trusted authorization server and that it has not been altered by either the client or any other party on the network.

It is important to note that the `/authz-info` endpoint of the resource server is not protected by any security scheme, given that said endpoint allows the client and resource server to establish a secure channel in the first place.

If the resource server verified the authenticity and integrity of the access token, it inspects the token's `aud` and `scope` parameters to check whether the requested audience matches the audience assigned to this resource server and whether the requested scopes are valid with respect to the requested resource.

The process of uploading access tokens prior to requesting the protected resource is in direct contrast to how access tokens are used in OAuth 2.0. In OAuth, the access tokens

are sent along with every request to a protected resource, usually as an HTTP header. In the ACE framework, clients interested in accessing protected resource on a resource server would upload their respective access token only once, assuming that the tokens are still valid. This further reduces the message size of the requests to the protected resources.

Resource servers running on constrained devices have to work with a potentially very limited amount of memory such that it may not be possible for the resource server to store multiple access tokens due to these memory restrictions. As a consequence, the ACE framework requires a resource server implementation to store at least one access token [1, p. 33]. While our Python resource server implementation can store multiple access tokens, the embedded resource server implementation is only able to store a single access token. Even though the platform our embedded resource server is running on would allow for multiple tokens to be stored, storing a single token is closer to the behavior of an implementation running on an even more constrained device. As a result of only being able to store a single token, the resource server is unable to maintain concurrent communication sessions with multiple clients. The clients would need to access the resource on the constrained resource server in succession, uploading their respective access token prior to the resource request.

4.5.6.1 Resource Request

After the access token has been uploaded to the resource server by the client, both parties have established the cryptographic keys necessary to establish a mutually authenticated and secure channel necessary for accessing the protected resource. We have illustrated the state of knowledge about the client's and resource server's keys in Figure 4.5.

Fig. 4.5: Keys established after token upload. The green keys have been established as part of the protocol flow up to the point where the access token has been uploaded to the resource server. The blue keys have been provisioned prior to the protocol flow.

After uploading the token to the resource server, the client is able to authenticate messages from the resource server using the resource server's public key – $ID_{RS,pub}$ – which the client has received from the authorization server as part of the response for the initial access token request. The client can confirm the authenticity of $ID_{RS,pub}$ since it has received said key over a secure channel with a trusted authorization server. Furthermore, the resource server is able to authenticate messages from the client using the proof-of-possession key generated by the client, $K_{PoP,pub}$.

The fact that both parties can authenticate each others messages allows them to establish a secure context. In the implemented security profile, this secure context is composed of a

symmetric key and initialization vector which will then be used to symmetrically encrypt the request and response to the protected resource. The client and resource server will only be able to agree on a key if the client can demonstrate the possession of the private proof-of-possession, $K_{PoP,prv}$, during the establishment of the secure context. We discuss the establishment of this secure context in the following chapter.

Once the client and resource server have established a secure context, communication between the client and resource server is confidential. Thus, the client performs the actual request to the protected resource at the appropriate URI. The client will encrypt the parameters of the request using the security context obtained in the previous step.

Upon reception of the request to the protected resource, the resource server use its security context to decrypt the request parameters and retrieve the token that was previously uploaded by the client. The resource server then checks whether the client is allowed to access the resource at the requested URI by inspecting the `scope` parameter in the access token. If the client is approved to access the resource, the resource server returns the resource encrypted using the established security context, which the client will be able to decrypt using its security context.

4.5.6.2 Introspection

The access token depicted in Section 4.5.3.1 are self-contained, meaning that the access token's payload contains all the authorization information associated with that access token. The benefit of self-contained tokens is that the resource server can inspect all the authorization claims contained in the access token without the help of any third party.

In some cases, however, the resource server is a highly constrained device incapable of inspecting access tokens. For such devices, the ACE framework proposes an alternative protocol flow where the access token issued by the authorization server is a simple string or number referencing the authorization information stored on the authorization server. With the authorization claims stored on the authorization server, whenever a client uploads a referential token to the resource server, the resource server will *introspect* the access token with the help of the authorization server. This introspection flow is also depicted in Figure 4.1. Before sending a response back to the client, the resource server sends the referential token to the `introspect` endpoint of the authorization server. The authorization server retrieves the authorization claims associated with the provided reference and checks whether the claims are valid for the requesting resource server. If the token is valid and active, the authorization server responds with a successful message and also includes the proof-of-possession key that the resource server will use to authenticate the client during the establishment of a secure context.

In our implementation, we only have limited support for referential tokens. While we have an implementation for the `introspect` endpoint at the authorization server, there is currently no possibility to have the authorization server issue referential tokens.

5
Security Profile

In this chapter, we take a detailed look at the security profile we implemented to secure the communication between the client and resource server while accessing a protected resource.

5.1 Requirements and Considerations

As stated in Chapter 3, the OAuth 2.0 specification requires certain interactions between the client, authorization server and resource to be secured by HTTPs. HTTPs is an application layer protocol which uses the transport layer security (TLS) protocol to encrypt messages exchanged between entities. In simple terms, the TLS protocol employs an asymmetric key exchange scheme, where the client and server agree on a symmetric key to be used for encryption. The public keys exchanged during the protocol are authenticated using digital certificates issued by certificate authorities as part of a public key infrastructure (PKI).

While this rigid approach to security works well in an environment where all communicating nodes are always connected to the Internet, and thus the PKI, and all nodes are computationally capable enough to support the cryptographic operations required in TLS, constrained nodes in IoT networks require more flexiblity with respect to the available processing power and their supported transport protocols.

The ACE framework addresses this need for flexibility and introduces the concept of *profiles* that can be implemented by the ACE entities according to their networking and processing capabilities. While the use of HTTPs is no longer mandatory, the ACE framework still requires these profiles to state how they protect communication between ACE entities. The following interactions all are required to be encrypted and integrity protected:

- The token request and response between client and authorization server
- The request and response to the protected resource between client and resource server

Additionally, all these interactions are required to be mutually authenticated. Mutual authentication for the token request is achieved, as stated in Section 4.5.1, by the client

presenting its client credentials to the authorization server, while the client authenticates the authorization server using the provisioned public key of the authorization server. Similarly, during the request to the protected resource, the client authenticates the resource server using the resource server's public key that the client received from the authorization server. The resource server authenticates the client as part of the proof-of-possession scheme, using the PoP key bound to the access token the client uploaded to the resource server. Due to this reason, an ACE profile is required to incorporate support for a proof-of-possession scheme.

At the time of writing this report, there are drafts for two profiles intended for use in an ACE authorization flow.

DTLS

The `Datagram Transport Layer Security` (DTLS) profile proposed by Gerdes et. al. in [17] uses the CoAP protocol secured by datagram transport layer security (DTLS). DTLS is an adaption of TLS designed to use UDP as its transport protocol instead of TCP. While very similar to HTTPs in OAuth2.0, the use of CoAP as the application layer protocol and DTLS as secure transport protocol reduces the message overhead and size significantly.

OSCORE

Proposed by Seitz et. al. in [16], the `Object Security for Constrained RESTful Environments` (OSCORE) profile is an application layer security protocol that provides end-to-end security between the communicating parties. As an application layer protocol, OSCORE is agnostic to the underlying transport protocol. While the specification in [16] is targeted to CoAP, OSCORE can be used with HTTP as well.

For this project, we chose to implement a security profile that builds upon the foundations of the OSCORE profile.

5.2 Object Security for Constrained Restful Environments (OSCORE)

OSCORE, as proposed in the IETF draft, was designed to address the shortcomings of DTLS in network deployment environments with proxy nodes. As the draft suggests in [16, p.4], these proxies are intended to increase efficiency and scalability, but require the security offered by DTLS to be suspended for the proxy operation. This poses a potential security risk as a compromised proxy node can inspect or even change messages sent to the proxy.

OSCORE offers full end-to-end security while still allowing proxy operations. For the CoAP protocol, it encrypts message payload as well as all options (which are the equivalent of HTTP headers) which are not relevant for proxy processing. While the OSCORE specification was written with CoAP as the underlying application layer protocol, in our project, we use OSCORE with HTTP instead of CoAP. We are not aware of any IETF drafts or specification that cover the use of OSCORE with HTTP. Thus, we opted to

adapt the core mechanisms introduced in OSCORE and implement a custom application layer security scheme suitable for HTTP.

5.2.1 Overview

The OSCORE protocol allows two communicating endpoints to establish a confidential channel where the messages exchanged between the endpoints are encrypted using a symmetric encryption scheme. As discussed in Section 2.3, symmetric encryption requires the two communicating parties to agree on a set of cryptographic parameters in order for the encryption scheme to be carried out successfully. While the OSCORE specification itself does not specify how the two parties establish the common keying material, it does suggest that constrained implementations use *Ephemeral Diffie-Hellman Over COSE*, or EDHOC. EDHOC is an asymmetric key exchange protocol that uses the COSE message format to derive cryptographic keys which can then be used to symmetrically encrypt OSCORE messages. In Figure 5.1, we illustrate how EDHOC is used to derive a set of parameters which are then used in the OSCORE protocol to actually encrypt the message exchange between a client and a server.

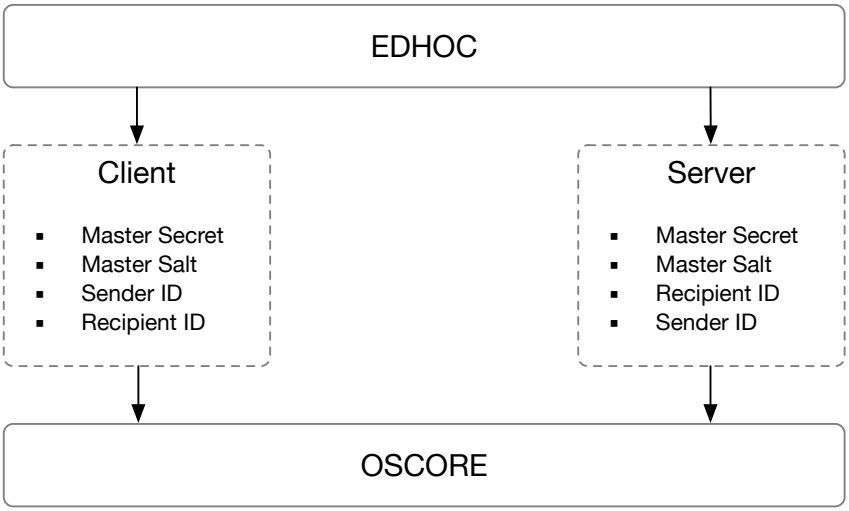

Fig. 5.1: Establishment of an OSCORE security context using EDHOC

In our implementation, we use EDHOC to establish a *master secret*, *master salt* and a respective *sender ID* and *recipient ID* for both the client and server. The master secret and master salt are shared by both parties, where as the sender ID and recipient ID are mutually mirrored for the client and server. These parameters will be used in the OSCORE protocol to derive content encryption and decryption keys which are then used to encrypt the message traffic between the client and server.

In the following section, we will further discuss how OSCORE secures traffic between a client and a server. We give a detailed account of how we use EDHOC to establish the cryptographic parameters required by OSCORE in Section 5.3.2

In order to understand the mechanisms and parameters used in OSCORE, we deem it helpful to first introduce two cryptographic algorithms we used in our implementation.

Advanced Encryption Standard The *Advanced Encryption Standard* or `AES` is a symmetric block cipher used to encrypt a plaintext byte string with the goal to completely transform said plaintext into an obscure ciphertext. The operations carried out by AES can be written as

$$C = \text{AES}(P, K, IV) \quad \text{(Encryption)} \tag{5.1}$$
$$P = \text{AES}(C, K, IV) \quad \text{(Decryption)} \tag{5.2}$$

where C is the ciphertext, P is the plaintext and K and IV are the encryption key and initialization vector (IV) respectively.

As a block cipher, AES is applied to the plaintext by separating the plaintext into equal sized blocks of 128 bits or 16 bytes. To encrypt a plaintext of arbitrary length, AES is coupled with a *mode of operation* which specifies how the cipher is applied to each block of the plaintext [22, p.124-135]. In our implementation, we use the `Counter with CBC-MAC` (CCM) mode of operation which provides us with *authenticated encryption* (AE). In addition to confidentiality, authenticated encryption also provides *authenticity* and *integrity* for the encrypted ciphertext. Thus, the receiver of the encrypted ciphertext C can verify that it has not been tampered with during transit on the communication channel and that C was sent by a party in possession of the encryption key K.

The CCM mode of operation, as a version of the general *counter* mode of operation, encrypts each block by applying the AES cipher on a combination of a random nonce and a successively increased counter value. The random nonce is set to the *initialization vector* (IV) value passed to the AES function. For each block, the counter is increased to the next value and combined with the initialization vector. This combined value is then used to encrypt each plaintext block to produce the final ciphertext.

Since the CCM mode of operation turns the application of AES into a stream cipher [4, p. 233, Remark 7.25], AES in CCM mode is vulnerable to attacks where the same key stream is used more than once to encrypt different plaintexts. The key stream is determined by the encryption key K and the initialization vector IV. If an attacker gets access to ciphertexts which were encrypted with the same key-IV pair (K, IV) and thus the same key stream, the attacker will be able to correlate the plaintexts without having to compute the encryption key [5, p. 255, Section 11.2.1]. As a consequence, using the CCM mode of operation with a constant encryption key K, every message to be encrypted must be encrypted with a different initialization vector. For this reason, initialization vectors are often also called a *nonce*. It is important to note that in contrast to the encryption key K, the initialization vector is not secret and can be transported in the clear.

Key Derivation Function A *key derivation function* or *KDF* is a cryptographic operation which allows keys to be derived from other keys. Key derivation functions are often used to derive strong cryptographic keys from short passwords or weak keys in order to increase the time spent by attackers in simple brute-force attacks.

In OSCORE, a key derivation function called `HKDF`, or `HMAC` based KDF is used to derive encryption and decryption keys from a master secret. A key derivation using HKDF can be formalized as

$$K_o = \text{HKDF}(K_i, L, S, \text{info})$$

where K_i is the input keying material, K_o is the derived key, S is a random byte string called *salt* and `info` is application specific byte string. The `salt` parameter is similar in function to the `IV` parameter in `AES` in that it is used to strengthen the output keying material. However, while an initialization vector must only be used once, the `salt` parameter used in HKDF can be reused multiple times. The `info` parameter is an additional source of entropy for the output keying material and is usually a byte representation of application specific data related to the derived key.

5.2.2 Protocol

The OSCORE protocol provides confidential communication between two endpoints. It employs a symmetric encryption scheme where the encryption and decryption keys together with other required cryptographic parameters are derived from a *security context*. We have depicted the composition of the security context in Figure 5.2.

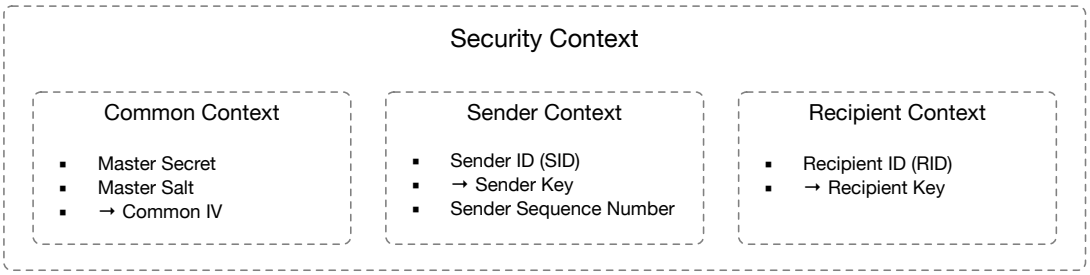

Fig. 5.2: Composition of an OSCORE security context

The security context is comprised of a *common context*, which is shared by both OSCORE endpoints. This common context is used by the respective communication parties to derive a *sender context* and *recipient context*, which will then be used to encrypt and decrypt messages exchanged in the OSCORE protocol using a symmetric encryption algorithm.

It is important to note that both OSCORE endpoints, henceforth called client and server, have both a sender and recipient context. In order for symmetric encryption to succeed, the recipient context of the server has to match the sender context of the client, and vice versa.

In OSCORE, the client and server both encrypt their messages using their respective sender context. To decrypt the received messages, the receiving party must derive an appropriate recipient context matching the sender context used by the other party to encrypt the message. To this end, we assume that both the client and server have been

assigned their respective `sender ID` (`SID`) parameter which is part of the endpoint's sender context.

We have depicted the protocol in Figure 5.3.

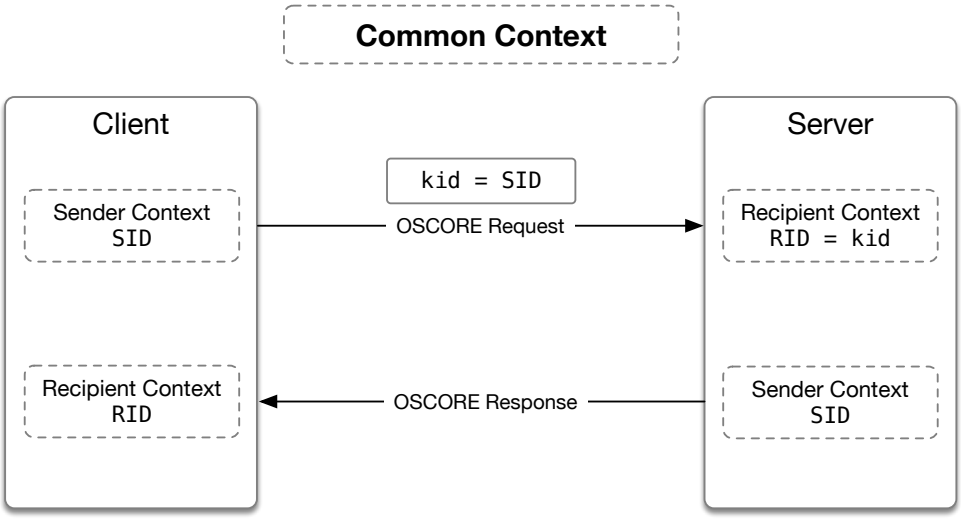

Fig. 5.3: OSCORE Message exchange protocol

The client encrypts its raw request with its sender context and sends the resulting encrypted OSCORE request to the server. Attached to the encrypted message, the sender also includes its sender ID `SID`. Upon reception of the OSCORE request, the server extracts the `SID` parameter from the request. At the server's side, the client's `SID` becomes the server's recipient ID `RID`. The server retrieves the security context which corresponds to the extracted `RID` and derives a recipient context which can then be used to decrypt the incoming request from the client.

In order to send a response, the server encrypts its raw response body using a sender context derived from its sender ID SID for the given client and transmits the resulting OSCORE response to the client. To conclude the protocol, the client uses its recipient context for the server and decrypts the OSCORE response.

The security contexts mentioned in the above mentioned protocol are composed of a certain set of cryptographic parameters required to encrypt or decrypt an OSCORE message, which we explain in the following section.

5.2.2.1 Security Context

Common Context The common context consists of a set of parameters shared by the sender and recipient of an OSCORE message. In our implementation, this common context consists of the following parameters:

- master secret (henceforth referred to as K_{MS})
- master salt (henceforth referred to as S_{MS})
- common initialization vector (common IV, henceforth referred to as $\mathrm{IV}_{\mathrm{common}}$)

The master secret K_{MS} and master salt S_{MS} are used to derive parameters for both the sender and recipient context. Additionally, the master secret and master salt are used to derive a common initialization vector (IV) which is also part of the common context.

As mentioned in Section 5.2.1, OSCORE does not specify how the master secret and master salt are established between the two parties. In our implementation, the common context is derived as part of the EDHOC key exchange protocol utilizing a proof-of-possession scheme facilitated by the access token. We give more details on the implemented key exchange protocol in Section 5.3.2.

We should note that the OSCORE profile specification lists many more parameters that are part of the common context, such as the specific encryption algorithm to use or the type of function used to derive keys. However, in our implementation, these parameters are implicitly part of the common context, since all parties use the same encryption algorithm and key derivation function for OSCORE related operations.

Sender Context The sender context consists of the following parameters:

- sender ID
- sender key
- sender sequence number

The `sender ID` or `SID` parameter identifies the sender in an OSCORE protocol. This parameter can provisioned by a mediating party such as an authorization server, or alternatively, the `sender ID` is established as part of the key exchange protocol executed to create the common context.

The `sender key` is a symmetric encryption key the sender uses to encrypt all its OSCORE messages. It is derived from the common context and `sender ID` parameter by using a key derivation function.

$$K_S = \text{HKDF}(K_{\text{MS}}, L, S_{\text{MS}}, \text{info})$$

where K_S is the derived sender key, K_{MS} is the master secret of the common context, L is the desired length of K_S, S_{MS} is the master salt and info is an the byte string of an encoded CBOR array containing parameters specific to the derived key [15, p. 11]. To derive a 16 byte sender key, we would have `info = [SID, 10, "Key", 16]` with the number 10 denoting the identifier of the encryption algorithm.

As previously discussed, using the same encryption key multiple times for different messages can result in the plaintext being retrieved from the ciphertext. To counter this attack vector, the sender generates a new initialization vector for each encrypted OSCORE message sent to the recipient. To generate this initialization vector, the sender keeps track of a `sender sequence number` which is increased by the sender with every OSCORE message sent to the recipient. The sender sequence number combined with the master salt from the common context and the sender ID of the sender context to generate a unique nonce that will be used as an initialization vector for every encrypted OSCORE message. Assuming a sender sequence number of n, the initialization vector would be

$$\mathrm{IV}_n = (\texttt{len(id)} \,\|\, \texttt{id} \,\|\, n) \oplus \mathrm{IV}_{\mathrm{common}}$$

where `id` is the sender ID, `len(id)` is the byte length of the sender ID and $\mathrm{IV}_{\mathrm{common}}$ is the common IV from the common context [15, p. 24]. The $\|$ symbol denotes byte string concatenation, and the $\oplus$ operator denotes a byte-wise `xor` operation. For brevity, we have omitted the necessary padding operations on some of the parameters.

Once the sender key K_S and IV_n have been computed, the sending party can encrypt its OSCORE message.

Recipient Context In order for the receiving party to decrypt the message, it first has to retrieve a recipient context associated with the sending party. The recipient context is used by an OSCORE endpoint to decrypt an OSCORE message. It comprises the following parameters:

- recipient ID
- recipient key

The *recipient ID* or `RID` identifies the recipient in the OSCORE protocol. Similar to the sender ID parameter of the sender context, this parameter is established using the EDHOC key exchange mechanism executed prior to the OSCORE protocol. The *recipient key* is the symmetric key derived from security context used to decrypt an OSCORE message. It can be derived from parameters of the other contexts as follows:

$$K_R = \mathrm{HKDF}(K_{\mathrm{MS}}, L, S_{\mathrm{MS}}, \mathrm{info})$$

where K_R is the derived recipient key and `info = [RID, 10, "Key", 16]` encoded as a CBOR array [15, p. 11]. It is evident that the sender's encryption key K_S and the recipient's decryption key K_R are the same as long as the `SID` of the sender's sender context matches the `SID` of the recipient's recipient context.

To decrypt the OSCORE message, the recipient also has to derive the initialization vector used by the sender in the encryption process. To that end, the recipient also computes

$$\mathrm{IV}_n = (\texttt{len(id)} \,\|\, \texttt{id} \,\|\, n) \oplus \mathrm{IV}_{\mathrm{common}}$$

where n is the sender sequence number and `id` is the recipient ID[15, p. 24]. The sender sequence number is not part of the recipient context, so the sender has to include said number as an unencrypted parameter in the request.

Once the recipient has derived the decryption key K_R and the initialization vector IV_n, it is able to decrypt the OSCORE message sent by the sending endpoint.

5.2.2.2 OSCORE Message

OSCORE uses the cryptography primitives provided by the COSE standard. As such, the encrypted OSCORE message uses the COSE `Encrypt0` structure as a wrapper for the encrypted data, which is a CBOR encoded array containing three elements:

```
[
  b'' // zero length map,
  {
    kid: sender ID,
    piv: sender sequence number
  },
  ciphertext
]
```

List. 5.1: OSCORE message encoded as CBOR array

As indicated in the previous section, the encrypted OSCORE message must include the sender's sender ID and sender sequence number in order for the recipient to be able to retrieve an appropriate security context capable of decrypting the encrypted payload. It is important to note that both of these parameters are sent as values in the unprotected header field of the COSE Encrypt0 structure, meaning that these parameters are not encrypted and therefore can be read without knowing the decryption key. If these parameters were encrypted, the recipient would not be able to retrieve a matching security context.

While the `kid` and `piv` parameters are sent unencrypted, we still need to make sure that these values are not tampered with by an attacker. For this reason OSCORE relies on the symmetric encryption algorithm to provide integrity as well as confidentiality. This is achieved by using an authenticated encryption cipher which offers the possibility to supply additionally authenticated data (AAD) to the encryption algorithm. These authenticated data will be used to produce a message authentication tag that is appended to the encrypted ciphertext. The authentication tag can then be used by the decrypting party to detect whether the ciphertext itself or the additionally authenticated data transmitted along with the ciphertext have been changed by an attacker on the communication medium.

When OSCORE is used with CoAP as defined in [15], the plaintext to be encrypted covers almost all of the CoAP options, which are similar to HTTP headers. Thus, even the URL of the request or the request method are integrity protected or even encrypted. In our implementation, where we use HTTP instead of CoAP, we have decided to encrypt solely the payload included in a request or response. In contrast to OSCORE with CoAP, this results in the URL path, HTTP method code and other potentially sensitive header values to be sent in plain text. Attackers on the same medium may be able to read and alter these values.

5.3 Ephemeral Diffie-Hellman over COSE (EDHOC)

The OSCORE profile specification does not specify how endpoints establish the required parameters of their respective security context. In small networks, where the number

of client-server combinations is limited, it may be feasible to pre-establish these security contexts. However, in networks where many clients can connect to a large number of servers, it is preferable to establish security contexts dynamically.

For this reason, the OSCORE specification suggests that implementations for constrained devices use an asymmetric key exchange protocol based on the *Diffie-Hellman* key exchange protocol. The Diffie-Hellman protocol allows two communicating parties to agree on a symmetric key, or shared secret, only by exchanging the public part of asymmetric keys.

Ephemeral Diffie-Hellman over COSE (EDHOC), as proposed by Selander et. al. in [14], is a protocol designed for constrained devices and implements a Diffie-Hellman key exchange where the messages exchanged between the parties are encoded as COSE objects. EDHOC employs an *elliptic curve Diffie-Hellman* (ECDH) key exchange scheme, meaning that the keys exchanged during the protocol are points on an elliptic curve. In the following section, we state how two communication parties can derive a shared secret using elliptic curve cryptography.

5.3.1 Diffie-Hellman Key Exchange with Elliptic Curve Cryptography

As a form of asymmetric cryptography, elliptic curve cryptography is based on the mathematical properties of a finite group of elements spanned by points on an elliptic curve along with an additive operation. Points P on an elliptic curve satisfy the curve equation $y = x^3 + ax + b$, where x and y are the coordinates of point P and a and b are parameters defining the shape of the elliptic curve . Along with other properties, points on an elliptic curve can be added to each other such that the resulting point is again a point on the elliptic curve [22, p. 242]. Additionally, a scalar multiplicative operation is defined where a point P on a curve is repeatedly added to itself, formally

$$\overbrace{P + P + P + \cdots + P}^{n \text{ times}} = nP$$

The security of the elliptic curve cryptosystem is built on the observation that for a chosen value of n with $Q = nP$, where Q and P are points on an elliptic curve, it is computationally very expensive to retrieve n if Q and P are known and n is very large [22, p. 247].

The abovementioned properties of points on an elliptic curve allow us to implement a Diffie-Hellman key exchange using elliptic curve points.

Suppose two parties A and B want to establish a shared secret s using elliptic curve cryptography. To this end both parties first have to agree on a set of parameters such that both parties use the same elliptic curve. This involves agreeing on a generator G, which is a point on the elliptic curve which can be used to generate any point on the curve. In practice, there is a large amount of standardized pre-defined curve parameters which implementations can choose from.

Once A and B agree on the curve parameters to use, both parties compute a key-pair (Q, d), where d is the private key, Q is the public key and $Q = dG$. Each party chooses a large respective d parameter randomly, such that party A ends up with the key pair (Q_A, d_A) and B with (Q_B, d_B). Both parties are required to keep their randomly chosen value of d_A or d_B secret.

To compute a shared secret s, A transports its public key Q_A to B, whereas B sends its public key Q_B to A. A computes the shared secret by computing $s_A = d_A Q_B$ and B by computing $s_B = d_B Q_A$. Due to the associative property of scalar multiplication, we have $s_A = d_A Q_B = d_A d_B G = d_B d_A G = d_B Q_A = s_B$. The x-coordinate of s_A or s_B respectively is then used as the shared secret s of parties A and B [22, p. 251].

The security of the above mentioned protocol can be further enhanced by generating a new asymmetric key pair for every session instead of using a single static key pair. The use of such *ephemeral* key pairs for every session assures that even if an attacker is able to obtain a secret key from one session, all other sessions will not be compromised. This property is known as *perfect forward secrecy* (PFS).

As stated above, computing d from Q and G with $Q = dG$ is computationally unfeasible. As a consequence, it is highly unlikely that any other party will be able to retrieve the private keys d_A and d_B used by A and B respectively. Furthermore, retrieving the shared secret $s_{A,B} = d_A d_B G$ from the exchanged public keys $Q_A = d_A G$ and $Q_B = d_B G$ is assumed to be as computationally expensive as the problem stated above [22, p. 251].

As in any asymmetric key exchange protocol, the public keys of A and B are required to be authentic in order to mitigate man in the middle attacks. To this end, the EDHOC specification extends the key exchange protocol explained above with public key authentication which ties into the proof-of-possession mechanism facilitated by the ACE access token. We explain this protocol in the following section.

5.3.2 EDHOC Message Exchange for Key Establishment

We explained in Section 5.2.1 and Figure 5.1 how EDHOC is used to derive a security context for both a client and a server in order for the two parties to be able to communicate with each other over a channel secured by symmetric encryption. In this section, we explain how EDHOC ties into the proof-of-possession scheme setup between an ACE client and resource server and how it establishes a master secret and master salt for the OSCORE security context which will then be used to secure the resource access from the client to the resource server.

Prior to making a request to a protected resource, the client has requested an access token for that protected resource at the resource server. In doing so, the authorization server issuing that token has bound a proof-of-possession key $K_{\text{PoP,pub}}$ to the access token. The client is the only entity in possession of the private key $K_{\text{PoP,pub}}$ associated with the proof-of-possession key.

After uploading the access token to the resource server, the resource server knows the proof-of-possession key $K_{\text{PoP,pub}}$ as well. Furthermore, the client knows the resource server's static public key $\text{ID}_{RS,pub}$ which it has received from a trusted authorization server as part of the response the access token request. The idea of the EDHOC protocol

is to use these two keys, $K_{\text{PoP,pub}}$ and $\text{ID}_{RS,pub}$, to authenticate the ephemeral public keys exchanged as part of the Diffie-Hellman protocol. As a consequence, the key exchange protocol will only succeed as long as the client can be authenticated by proving that it is in possession of the private key associated with the proof-of-possession key $K_{\text{PoP,pub}}$ bound to the access token.

In Figure 5.4, we depict the protocol flow between an ACE client and resource server, after the client has uploaded its access token to the resource server. The EDHOC protocol involves the exchange of three messages.

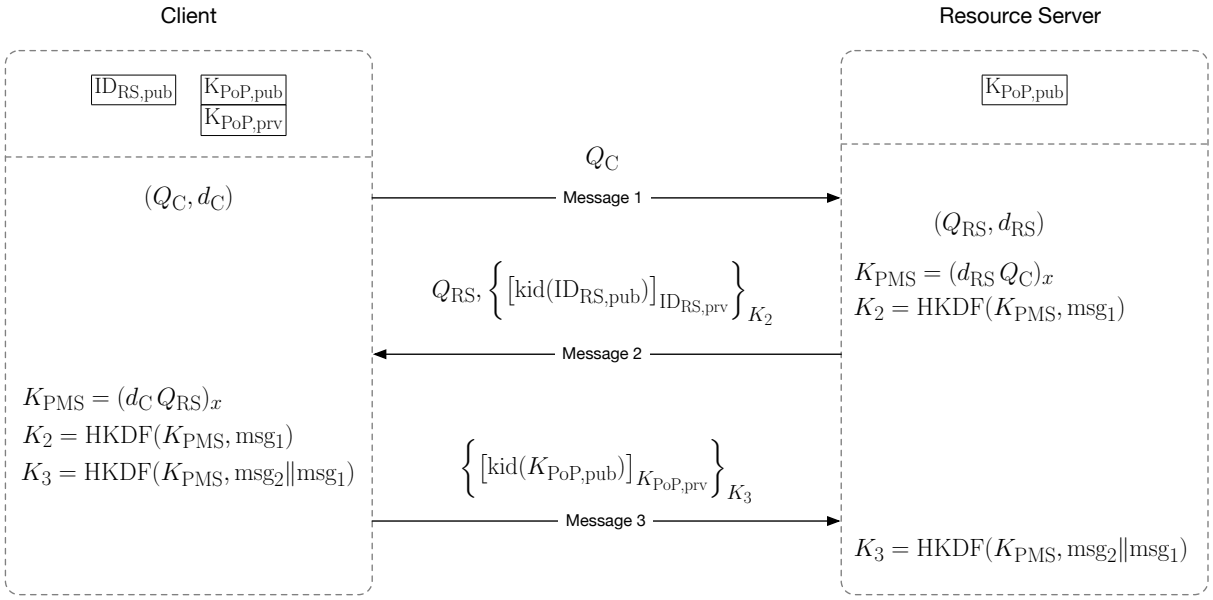

Fig. 5.4: EDHOC protocol flow, $\{\}_K$ denotes encryption with key K, $[]_K$ denotes digital signature with key K. For brevity, we have omitted the additional authenticated data and session parameters. Figure adapted from [14, p. 8]

As described in Section 5.3.1, the EDHOC protocol commences with the client generating an ephemeral key pair (Q_C, d_C) by computing $Q_C = d_C G$ where G is the generator of the curve agreed upon by both endpoints of the protocol [14, p. 10]. It then sends the public part Q_C to the resource server over a still unprotected channel. After the resource server has received the client's ephemeral public key Q_C, it also generates a key pair $(Q_{\text{RS}}, d_{\text{RS}})$ by computing $Q_{\text{RS}} = d_{\text{RS}} G$, where G is the same generator point used by the client.

At this point, the resource server is able to compute a *pre-master secret* (PMS) K_{PMS} by computing $K_{\text{PMS}} = (d_{\text{RS}} Q_C)_x$. From the pre-master secret, the resource server derives an encryption key K_2 using $K_2 = \texttt{HKDF}(K_{\text{PMS}}, \text{msg1})$. In this context, msg1 represents a byte string derived from the contents of the first received EDHOC message and is passed as the `info` parameter of the `HKDF` key derivation algorithm [14, p. 7]. To form EDHOC message 2, the resource server computes a digital signature for the key identifier of its static public key $\text{ID}_{\text{RS,pub}}$. Since the client may have multiple public keys for different resource servers, the client may use this identifier to retrieve the correct public key which it then uses to cryptographically verify EDHOC message 2. The digital signature also encompasses the resource server's computed ephemeral public key Q_{RS} by including it as additional authenticated data in the signature generation process. This digital signature

is then symmetrically encrypted with K_2 and sent to the client along with the resource server's ephemeral public key Q_{RS}.

As soon as the client has received EDHOC message 2, it needs to derive the decryption key K_2 in order to decrypt the received message. To that end, it first computes the pre-master secret by extracting Q_{RS} from the received message and computing $K_{\mathrm{PMS}} = (d_C Q_{\mathrm{RS}})_x$. The client also computes K_2 in the same manner as the resource server by deriving it from the pre-master secret. Once the client has computed K_2, it can decrypt the received message to obtain the digital signature $[\mathrm{kid}(\mathrm{ID}_{\mathrm{RS,pub}})]_{\mathrm{ID}_{\mathrm{RS,prv}}}$. The client extracts the key identifier for the resource server's static public key and retrieves the associated public key $\mathrm{ID}_{\mathrm{RS,pub}}$. Since the client has received this public key prior to the start of the EDHOC protocol, it can verify the signature and thus authenticate the received ephemeral public key Q_{RS}. From this point on, the server has confirmed that it is communicating with the resource server in charge of the resource covered by the access token.

As a last step, the client must authenticate itself to the resource server. To this end, the client derives another encryption key K_3 from the pre-master secret. It generates a digital signature of the key identifier of the proof-of-possession key $K_{\mathrm{PoP,pub}}$. This key identifier was chosen by the client when it generated the proof-of-possession key and is included in the access token uploaded to the resource server. Since the resource server may store multiple access tokens, the resource server will use the key identifier to extract the correct proof-of-possession key for that particular client. The client then encrypts the obtained digital signature using K_3 to form $\left\{[\mathrm{kid}(K_{\mathrm{PoP,pub}})]_{K_{\mathrm{PoP,prv}}}\right\}_{K_3}$, which then sent to the resource server as part of the last EDHOC message.

Once the resource server has received EDHOC message 3, it also derives the decryption key K_3 from the pre-master secret in the same manner as the client. It then decrypts the received message to obtain the digital signature $[\mathrm{kid}(K_{\mathrm{PoP,pub}})]_{K_{\mathrm{PoP,prv}}}$. The resource server extracts the key identifier from the message and retrieves the access token containing a proof-of-possession key matching the supplied key identifier. The resource server then extracts the proof-of-possession key $K_{\mathrm{PoP,pub}}$ from the access token and uses this key to verify the signature of the received third message. If the verification is successful, the client has demonstrated that it is in possession of the private key associated with the proof-of-possession key bound to the access token and is considered authenticated to the resource server.

It is important to note that the EDHOC protocol is terminated as soon as either party detects that any parameters exchanged over the communication channel has been altered by an attacker. Any alterations to the exchanged messages during the EDHOC protocol will result in either party not being able to derive the same encryption or decryption key, upon which the protocol fails and is terminated[14, p. 24].

OSCORE Security Context Parameters After a successful run of the EDHOC protocol described in the section above, the client and resource server end up with a pre-master secret K_{PMS} and an identifier for both the client and the resource server. To build a complete OSCORE security context as described in Section 5.2.2, the client and resource server both derive a master secret K_{MS} and master salt S_{MS} from the pre-master secret. At this point, both the client and resource server have established an OSCORE secu-

rity context capable of encrypting and decrypting messages exchanged while accessing a protected resource on the resource server.

6 Implementation

In this chapter we give an insight into our implementations of the various protocols and entities defined in the ACE framework.

6.1 Considerations

As of the time of writing this report, the only implementation of a complete protocol flow as proposed in the ACE framework is a work-in-progress implementation maintained by some of the authors of the ACE framework[1]. Their work covers implementations for the authorization server and resource server and are written in the JAVA programming language. Being based in the JAVA ecosystem, the authors can rely on the existence of a large number of libraries available to them, such as libraries for CBOR, COSE and CoAP. However, JAVA programs are executed inside of the JAVA runtime environment which is a virtual machine designed to make the execution of programs independent of the underlying hardware and operating system.

While executing JAVA programs is effortless for reasonably powerful devices such as smartphones, constrained nodes found in IoT networks are often far less powerful and may not even be able to run the JAVA virtual machine. For that reason we have set one of the goals of this project to implement a resource server on a code basis that is able to run on very constrained devices.

In order to decide on the feasibility of an implementation for constrained devices, we have concluded that it is helpful to first write an implementation of all ACE entities (authorization server, resource server, client) in the Python programming language. Given that there already are a lot of OAuth 2.0 implementations written in Python, and considering that the ACE framework itself is an extension of OAuth 2.0, we think that a Python implementation could provide a valuable alternative to the JAVA solution proposed by the ACE authors. The Python implementation also provided us with a lot experience which we could then profit from when developing the embedded implementation. The source code and a small setup guide for the Python implementation can be found on GitHub[2].

[1] https://bitbucket.org/lseitz/ace-java
[2] https://github.com/HappyEmu/ace

Artifacts As part of this project, we developed a set of three Python libraries all of which implement a certain set of protocols and entities required to cover a run of the ACE protocol flow. The libraries and their relations are depicted in Figure 6.1.

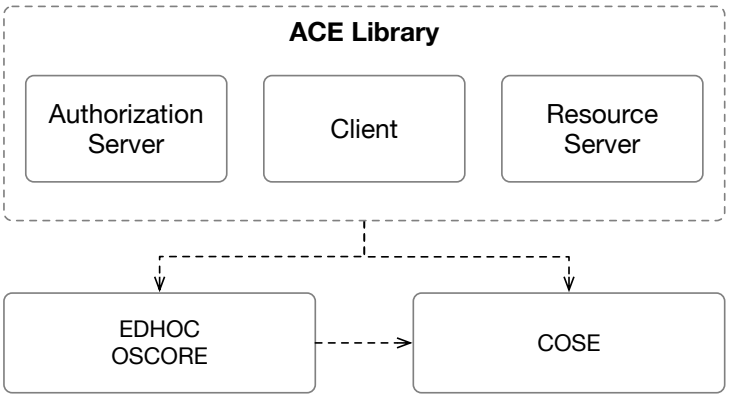

Fig. 6.1: Implemented libraries along with their dependencies on each other.

ACE Library
The ACE library consists of implementations for all three ACE entities, i.e. authorization server, resource server and client.

COSE Library
The COSE library models objects from the COSE standard. This includes CBOR encoded encryption objects and digital signature objects as well as COSE formatted keys. This library is used by the ACE library as well as the EDHOC library where appropriate.

OSCORE / EDHOC Library
The OSCORE / EDHOC library provide classes that implement the EDHOC key exchange and the subsequent OSCORE message exchange. This library uses COSE objects provided by the COSE library and is consumed by the ACE library in order to secure the messages exchanged between the client and resource server.

In Figure 6.2, we have depicted the internal structure of our implemented modules along with their dependencies on each other.

Third-Party Dependencies As is common with all software, our implementation uses some third-party libraries that have helped us achieve our goal. All cryptographic algorithms used throughout the ACE framework, such as AES encryption, elliptic key pair generation, key derivation functions and digital signature algorithms are supplied by the `cryptography` Python package, which is a collection of cryptographic primitives implemented in Python. More specifically, `cryptography` delegates to `OpenSSL`, which is the de facto standard for cryptographic computations. Additionally we use the `cbor2` library which provides us with the possibility to encode arbitrary Python objects as binary CBOR byte strings. Lastly, we use the `aiohttp` library which offers means and classes to write an asynchronous HTTP server.

In the following sections, we provide insight into the respective libraries developed for this project.

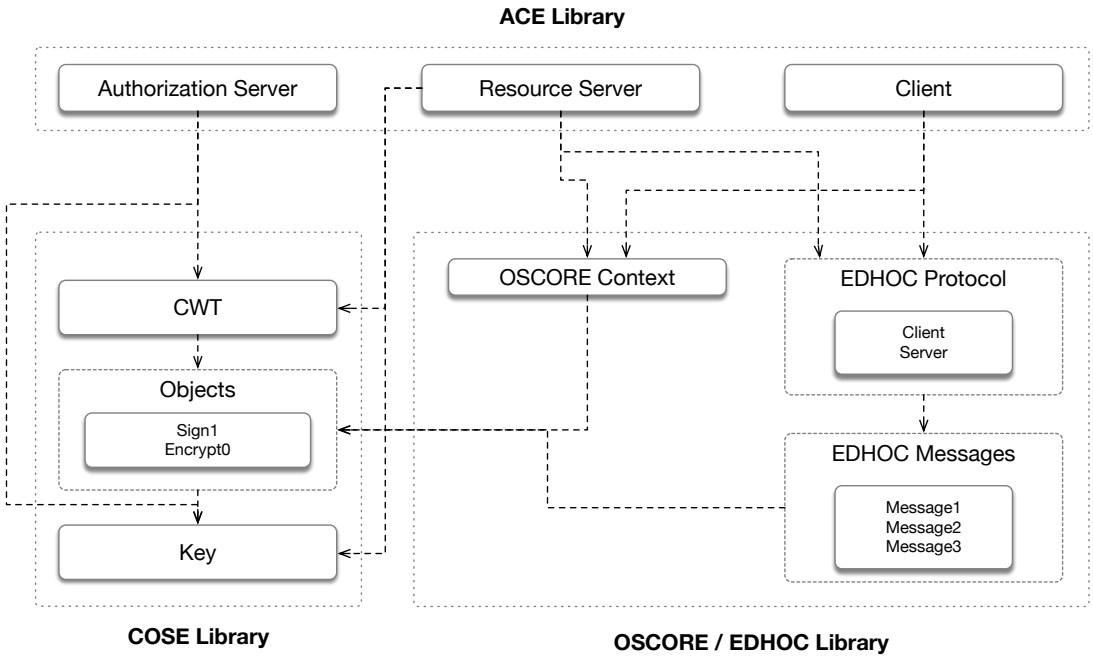

Fig. 6.2: Internal structure and implemented modules of the Python libraries.

6.2 ACE Library

Our Python implementation of the ACE framework includes classes for all three ACE entities, authorization server, resource server and client. In the following sections, we document how consumers of our libraries can setup a full ACE protocol flow by demonstrating the public programming interface (API).

6.2.1 Authorization Server

Consumers of our library can create a new ACE authorization server as illustrated in Listing 6.1.

```
# Provision private key of authorization server
as_identity = SigningKey.from_der(
    bytes.fromhex("3077[...]3355")
)

server = AuthorizationServer(identity=as_identity)

# Pre-register resource servers
server.register_resource_server(
    audience="tempSensor0",
    scopes=['read_temperature', 'post_led'],
    public_key=VerifyingKey.from_der(
        bytes.fromhex("3059[...]6d7a")
    )
)

# Pre-register clients
server.register_client(
```

```
    client_id="ace_client_1",
    client_secret=b"ace_client_1_secret_123456",
    grants=[
        Grant(
            audience="tempSensor0",
            scopes=["read_temperature", "post_led"]
        )
    ]
)

server.start(port=8080)
```

List. 6.1: Authorization server

The first step is to create a new instance of the `AuthorizationServer` class, which takes as an input parameter the private key the authorization should use to sign the access token it issues.

The next step involves provisioning all the resource servers that this authorization server controls. This can be done by calling the `register_resource_server(...)` method on the authorization server instance along with the audience and scopes this resource server provides. Additionally, the consumer has to provide the static public key of the registered resource server. The authorization server needs to know this key in order to provide a connecting client with the resource server's public key that it should use to authenticate messages from the resource server.

Furthermore, consumers need to register all clients that are allowed to access protected resources on the registered resource servers. This is done by invoking the `register_client(...)` method with the client's credentials and the grants that this client is allowed to access. The authorization server will check all incoming token requests against the registered clients to verify that only registered clients are allowed to request an access token.

As a last step, the now configured authorization server instance can be started by calling the `start` method and providing the port number the server should be bound to. From this point on, the authorization server is ready to respond to incoming token requests from registered clients.

6.2.2 Resource Server

Consumers can implement their own resource servers by extending the provided `ResourceServer` class. An example resource server with two protected scopes is illustrated in Listing 6.2.

```
class TemperatureSensor(ResourceServer):

    def on_start(self, router):
        super().on_start(router)

        router.add_get(
            '/temperature',
            self.wrap(scope="read_temperature", handler=self.get_temperature)
        )
        router.add_post(
            '/led',
```

```
            self.wrap(scope="post_led", handler=self.post_led)
        )

    # POST /led
    def post_led(self, request, payload, token, oscore_context):
        data = loads(oscore_context.decrypt(payload))

        print(f"Setting LED value to: {data[b'led_value']}")

        response = oscore_context.encrypt(dumps(b'OK'))
        return web.Response(status=201, body=response)

    # GET /temperature
    def get_temperature(self, request, payload, token, oscore_context):
        temperature = random.randint(8, 42)
        response = oscore_context.encrypt(dumps({'temperature': f"{temperature}C"}))

        return web.Response(status=200, body=response)
```

List. 6.2: Defining Resources on Resource Server

The binding from the resource's URL endpoint to the handler is defined in the `on_start` method, where consumers state how URLs are mapped to the scope and handler of a protected resource. The handlers will automatically be invoked with the appropriate OSCORE security context which can be used by the handlers to decrypt the payload and encrypt responses.

The defined resource server can then be started as shown in Listing 6.3

```
rs_identity = SigningKey.from_der(
    bytes.fromhex("3077[...]6d7a")
)

as_public_key = VerifyingKey.from_der(
    bytes.fromhex("3059[...]3355")
)

server = TemperatureServer(
    audience="tempSensor0",
    identity=rs_identity,
    as_url='http://localhost:8080',
    as_public_key=as_public_key
)

server.start(port=8081)
```

List. 6.3: Executing Resource Server

To start the resource server, consumers have to create an instance of their resource server implementation. The constructor of the `ResourceServer` class requires that consumers pass the audience the resource server should identify itself with. In order to perform introspection requests, consumers also have to pass the URL of the authorization server, as well as its public key which will be used by the resource server to authenticate signatures and access tokens issued by the authorization server. Furthermore, consumers are required to provide a static private key whose corresponding public key the other parties use to authenticate messages from the resource server.

As a last step, we can instruct the resource server start listening by calling the `start` method and passing the port number the underlying HTTP server should bind to.

6.2.3 Client

The `Client` class can be used to access protected resources on resource servers. It models all interactions with the authorization server as well as the final request to the resource server. Listing 6.4 shows an example where the authorization server and resource server started in the previous sections are queried.

```
AS_URL = 'http://localhost:8080'
RS_URL = 'http://localhost:8081'

client = Client(
    client_id='ace_client_1',
    client_secret=b'ace_client_1_secret_123456'
)

# Request access token
session = client.request_access_token(
    as_url=AS_URL,
    audience="tempSensor0",
    scopes=["read_temperature", "post_led"]
)

# Upload token to RS
client.upload_access_token(session, RS_URL, '/authz-info')

# Access temperature resource
response = client.access_resource(session, RS_URL + '/temperature')
print(f"Response: {response}")

# Update LED resource on RS
data = { b'led_value': 1 }
response = client.post_resource(session, RS_URL + '/led', dumps(data))
print(f"Response: {response}")
```

List. 6.4: Accessing Resources on Resource Server

Consumers of the ACE library can create a client by instantiating an instance of the `Client` class by providing the client credentials that should be used for the created client. The client will only be able to request access tokens from the authorization server if its credentials were previously provisioned when creating the authorization server object.

A new *session* is created as soon as the client requests an access token from the authorization server by calling the `request_access_token(...)` method on the `Client` instance. To said method, the consumer is required to pass the audience the client wants to access later as well as all scopes that should be covered by the returned access token. Additionally, the authorization server's URL needs to be supplied so the client can make the correct request. If the token request succeeds, the method returns a `session` object, which encapsulates all the state that is associated with the returned access token.

As a next step, consumers can actually perform a request to a protected resource by either calling the `access_resource(...)` or `post_resource(...)` method. The former will

issue a `GET` request while the latter will `POST` some data to the resource server. The session object referring to the obtained access token is used so these requests are performed in the correct context. The returned `response` will contain the decrypted response from the resource server for the accessed protected resource.

Example usage of the COSE and EDHOC libraries can be found in the software documentation.

6.3 Embedded Resource Server

In addition to the Python libraries mentioned in the sections above, we also provide a resource server implementation capable of running on a constrained embedded device. Our embedded resource server implementation is written in the `C` programming language and is built against the *Mongoose OS* IoT development platform[3]. Similar to the Python implementation, the source code along with a brief setup guide for the embedded resource server implementation can be found on GitHub[4].

Mongoose OS is composed of a set of tools aimed to facilitate fast development of firmware for IoT devices. As such, it builds upon the foundations of the *Mongoose* networking library[5] which provides means to build embedded web servers. We use the networking functionality provided by the *Mongoose* networking library, which is integrated into *Mongoose OS*, to implement our ACE resource server solution. To facilitate encoding CBOR objects, we have ported the `tinycbor` library provided by Intel to be compatible with Mongoose OS. Mongoose OS provides an integrated version of the `mbedtls` crypto library written by ARM which provides algorithms and data structures related to cryptographic operations such as encryption and digital signatures. Similar to the Python implementation, we have developed our own implementations of the COSE and CWT objects. Additionally, the embedded resource server also supports the establishment of an OSCORE security context using the EDHOC key exchange protocol.

We run the embedded resource server implementation on a *Widora AIR ESP32* development board which features an ESP32 system-on-a-chip microcontroller running at 140 megahertz. The ESP32 microprocessor has integrated Bluetooth and Wi-Fi support and features 520 kilobytes of on-chip random access memory (RAM). With these specifications, this device would not qualify as a constrained node as defined in [6] since it exceeds the capabilities of *Class 2* constrained nodes. However, as mentioned by the authors of [6], the classification should only be viewed as rough guidelines.

While the `mbedtls` library is capable of handling a wide variety of cryptographic operations using the processing capabilities of the ESP32 microcontroller, we aim to increase the throughput of the resource server implementation by offloading some of the cryptography related computations to a dedicated security chip.

[3]https://mongoose-os.com
[4]https://github.com/HappyEmu/ace-rs-mgos
[5]https://cesanta.com/docs/overview/intro.html

6.3.1 Secure Element

One challenge for authentication and authorization for devices in IoT networks is the fact that the network nodes may be physically accessible to malevolent parties. This poses a serious security risk since attackers may be able to retrieve the private key of asymmetric key pairs by inspecting the application running on the embedded device with appropriate tools. In our project, we counter this threat by using a *secure element* that is capable of performing cryptographic operations in isolation of the application and is designed to withstand tampering attempts. By using this secure element in asymmetric cryptographic operations, the private key never leaves the secure element and is not exposed to working memory of the application.

In our implementation, we use a *ATECC508A* crypto element built by Microchip. The ATECC508A supports elliptic curve cryptography with the `P-256` curve and has the capability to perform computations related to Elliptic Curve Diffie-Hellman (ECDH) which we cover in Section 5.3.1. Additionally, the crypto element provides means to digitally sign as well as verify signatures using the Elliptic Curve Digital Signature Algorithm (ECDSA). We use the secure element to compute the ECDH key exchange mechanism which is part of the EDHOC key exchange protocol. Furthermore, the embedded resource server delegates the verification of the access token's digital signature, as well as all COSE signature objects that are used during the EDHOC protocol flow, to the secure element. The application communicates with the ATECC508A crypto element over a two wire serial *I2C* protocol.

6.3.2 Assembly

For the embedded resource server, we emulate both a protected resource and an actuator associated with the resource server. Clients should be able to access the protected resource from the resource server, as well as update the value of the actuator by means of an ACE protocol flow. To this end, we have built an assembly where we connect a *DHT22* digital humidity and temperature sensor and an LED to the input and output ports of the Widora AIR board. The DHT22 is responsible for providing the protected temperature and humidity resource, where the LED represents a protected actuator value that can be enabled or disabled. We illustrate a schematic of the assembly in Figure 6.3 and an image of the actual assembly setup is shown in Figure 6.4

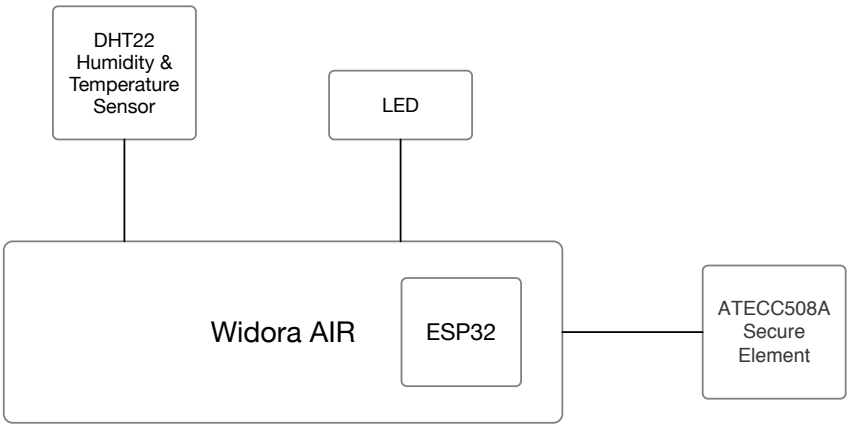

Fig. 6.3: Schematic view of the assembly of the embedded resource server

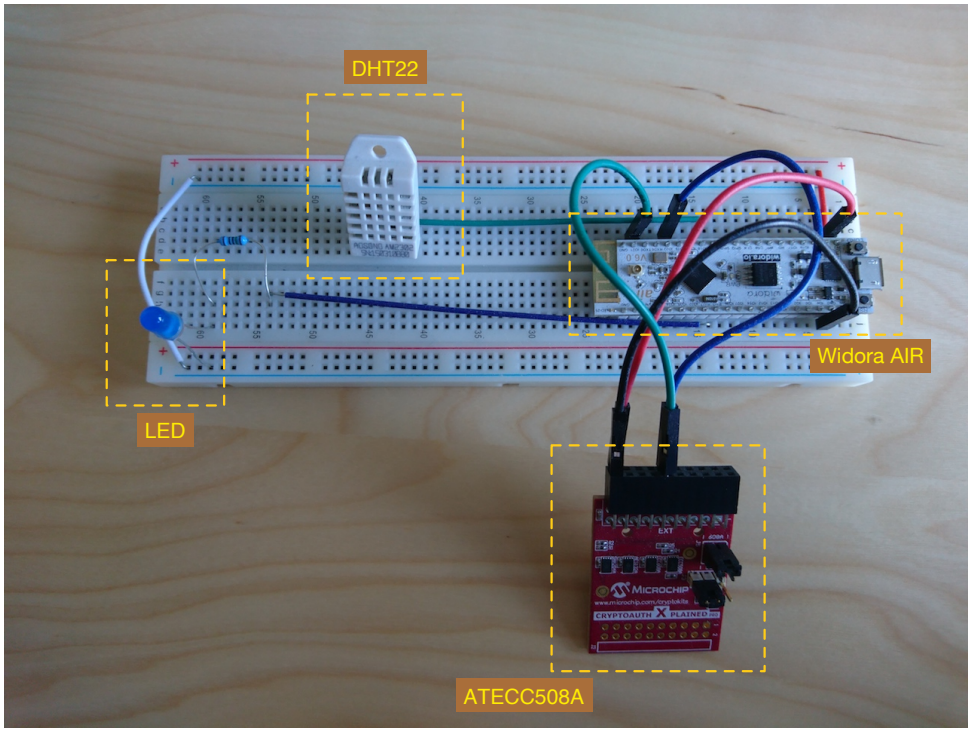

Fig. 6.4: Photograph of the assembled embedded resource server

7 Results

In this chapter, we present the results we obtained from our implementations. We depict the actual data that is exchanged during an ACE protocol flow while recording and reporting on the sizes of the messages being exchanged. We compare the obtained message sizes with messages exchanged in a traditional OAuth 2.0 protocol flow and provide some basic performance measurements to quantify the speedup gained from using a dedicated crypto element performing asymmetric cryptography computations.

7.1 Example Request

In this section, we illustrate and decode the actual binary data that are exchanged between ACE entities during an ACE protocol flow.

7.1.1 Token Request and Response

The client starts by sending a Token Request to the authorization server. The resulting HTTP request is illustrated in Listing 7.1. It is important to note that for the purpose of illustration, transport layer security for the communication between the client and authorization server has been disabled.

```
1  POST /token HTTP/1.1\r\n
2  Host: localhost:8080\r\n
3  Content-Length: 183\r\n
4  \r\n
5  [
6    a61202086c6163655f636c69656e745f3109581a6163655f636c69656e745f31
7    5f7365637265745f3132333435360c7819726561645f74656d70657261747572
8    652c706f73745f6c6564036b74656d7053656e736f72301819a101585aa50102
9    200121582020d6611e7097d1f0a8c1b8a5cd7f7fd60d089454130df26e613eb2
10   0c08f51e182258208f9a74ef40f470c39c3fd2f892551f171626b43074c28778
11   25e00eed81d4c4ed024d6163655f636c69656e745f3130
12 ]
```

List. 7.1: HTTP Request for Token Request

In this example, the client makes a `POST` request to an authorization server listening on the URL `localhost:8080`. The listing also shows how the client encodes the request parameters as a CBOR map resulting in a payload size of 183 bytes. The decoded payload is illustrated in the following listing.

```
{
   / grant_type /   18: 2, // 2 = "client_credentials"
   / client_id /     8: "ace_client_1",
   / client_secret / 9: h'6163655F636C69656E745F315F736563
                          7265745F313233343536',
   / scopes /       12: "read_temperature,post_led",
   / audience /      3: "tempSensor0",
   / cnf /          25: {
      / COSE_Key / 1: h'A501022001215820 20D6611E7097D1F0
                        A8C1B8A5CD7F7FD60D089454130DF26E
                        613EB20C08F51E182258208F9A74EF40
                        F470C39C3FD2F892551F171626B43074
                        C2877825E00EED81D4C4ED024D616365
                        5F636C69656E745F3130'
   }
}
```

List. 7.2: Token Request parameters

Listing 7.2 shows the decoded parameters that the client is sending to the authorization server as part of the initial Token Request. In the example, the client instructs the authorization server to use the *client credentials* flow and includes its client credentials `client_id` and `client_secret`. Additionally, we can see how the client wants to access the `read_temperature` and `post_led` scopes on the resource server that identifies itself with the audience `tempSensor0`. Lastly, we can also observe the proof-of-possession key that the client wants the authorization server to bind to the issued access token. This PoP key is encoded in the `cnf` parameter of the Token Request parameters. The decoded version of this key is shown in Listing 7.3.

```
{
  / kty /  1: 2, // 2 = "Elliptic Curve"
  / crv / -1: 1, // 1 = "P-256 Curve"
  / X /   -2: h'20D6611E7097D1F0A8C1B8A5CD7F7FD6
               0D089454130DF26E613EB20C08F51E18',
  / Y /   -3: h'8F9A74EF40F470C39C3FD2F892551F17
               1626B43074C2877825E00EED81D4C4ED',
  / kid /  2: h'6163655F636C69656E745F3130'
}
```

List. 7.3: Proof-of-Possesion key generated by the client

The PoP key generated by the client is the public part of an elliptic curve key pair as described in Section 5.3.1. As such, the `kty` (key type) parameter is set to a CBOR value that identifies the key as an elliptic curve key. The `crv` parameter is used to declare which elliptic curve this public key is an element of. Throughout our implementation, we use the widely supported `P-256` curve proposed by the National Institute of Standards and Technology (NIST) in [18]. As an elliptic curve public key, the PoP key is a point on the indicated elliptic curve and thus features both an x and y coordinate, which are also encoded in the COSE key structure of Listing 7.3. Lastly, the client has assigned

a key identifier such that other entities such as the resource server can retrieve the key associated with the given key identifier. This key identifier is visible as the `kid` parameter.

As soon as the authorization server has authenticated the client and verified the Token Request, it issues an access token valid for the requested scopes and returns a response as defined in Section 4.5.4 whose decoded content is illustrated in Listing 7.4.

```
{
   / access_token / 19: h'D28443A1012654A104516163652E6173
                           2D7365727665722E636F6D58AFA70171
                           6163652E61732D7365727665722E636F
                           6D061A5B643F16041A5B645B36076431
                           3731300C7819726561645F74656D7065
                           7261747572652C706F73745F6C656403
                           6B74656D7053656E736F72301819A101
                           585AA501022001215820 20D6611E7097
                           D1F0A8C1B8A5CD7F7FD60D089454130D
                           F26E613EB20C08F51E182258208F9A74
                           EF40F470C39C3FD2F892551F171626B4
                           3074C2877825E00EED81D4C4ED024D61
                           63655F636C69656E745F313058405581
                           81900163B37FDA76BF598AB51B43B84A
                           DD8692AC1F10A8211E5F6A3A113B4200
                           A13C6FCEB327861A2C096AEFDABB74A0
                           CA533A89CBC4EB982E7B04DB79B4',
   / token_type /   20: "pop",
   / profile /      26: "coap_oscore_edhoc",
   / rs_cnf /       31: h'A5010220012158206CC41512D92FB03C
                           B3B35BED5B494643A8A8A55503E87A90
                           282C78D6C58A7E3C22582088A21C0287
                           E7E8D76B0052B1F1A2DCEBFEA57714C1
                           210D42F17B335ADCB76D7A024A72735F
                           7075625F6B6579'
}
```

List. 7.4: Token Response parameters

The authorization server instructs the client to use the `coap_oscore_edhoc` security profile towards the resource server and indicates that the returned access token is a proof-of-possession token. The authorization server also returns the resource server's public key that the client must use to authenticate messages from the resource server. Most importantly, the response to the Token Request also includes the issued access token. As mentioned in Section 4.4.2, the access token is a CBOR Web Token (CWT). Listing 7.5 shows the decoded CWT.

```
[
  h'A10126',                                      // protected header
  h'A104516163652E61732D7365727665722E636F6D',    // unprotected header
  h'A701716163652E61732D736572766572              // payload
    2E636F6D061A5B643F16041A5B645B36
    0764313731300C7819726561645F7465
    6D706572617475726 52C706F73745F6C
    6564036B74656D7053656E736F723018
    19A101585AA501022001215820 20D661
    1E7097D1F0A8C1B8A5CD7F7FD60D0894
    54130DF26E613EB20C08F51E18225820
```

```
    8F9A74EF40F470C39C3FD2F892551F17
    1626B43074C2877825E00EED81D4C4ED
    024D6163655F636C69656E745F3130',
  h'558181900163B37FDA76BF598AB51B43          // signature
    B84ADD8692AC1F10A8211E5F6A3A113B
    4200A13C6FCEB327861A2C096AEFDABB
    74A0CA533A89CBC4EB982E7B04DB79B4'
]
```

List. 7.5: Decoded contents of the CWT access token

The decoded CWT shows the four elements of the underlying `COSE Sign1` structure. We can observe the digital signature computed by the authorization server using its static private key. This signature can be used by both the client and resource server to verify that this access token was indeed created by a trusted authorization server, whose public key they have been pre-configured with. The payload – the third element of the CWT – carries the authorization claims that the authorization server has bound to the access token. We illustrate the decoded payload in the following listing.

```
{
  / iss /     1: "ace.as-server.com",
  / aud /     3: "tempSensor0",
  / exp /     4: 1533303606,
  / iat /     6: 1533296406,
  / cti /     7: "1710",
  / scopes / 12: "read_temperature,post_led",
  / cnf / 25: {
    / COSE_Key / 1: h'A501022001215820 20D6611E7097D1F0
                       A8C1B8A5CD7F7FD60D089454130DF26E
                       613EB20C08F51E182258208F9A74EF40
                       F470C39C3FD2F892551F171626B43074
                       C2877825E00EED81D4C4ED024D616365
                       5F636C69656E745F3130'
  }
}
```

List. 7.6: Authorization claims included in the access token

In Listing 7.6 we can see the authorization claims that are included in the access token issued by our authorization server. The access token is valid for accessing protected resources from the resource server that is associated with the audience `tempSensor0` and is authorizing the client to perform actions covered by the scopes `read_temperature` and `post_led`. We should point out that the encoded proof-of-possession key that was bound to the access token in the `cnf` parameter of the access token is the exact same CBOR byte string that was supplied by the client as indicated in Listing 7.2.

7.1.2 Resource Access

In this section, we inspect the OSCORE messages that are exchanged after the access token has been uploaded to the resource server and the OSCORE security context has been established after completing the EDHOC key exchange protocol. The client was assigned a sender ID of `AC60` and a recipient ID of `42C7`. The respective IDs are mirrored

for the resource server. Since this is the first access to a protected resource, both the client's and the resource server's sender sequence number of their respective sender context is initialized to zero. In Listing 7.7 we have illustrated the payload sent by the client to the resource server.

```
[
  h'',                        // protected header
  {                           // unprotected header
    / piv / 6: h'00',
    / kid / 4: h'AC60'
  },
  h'2EA1E85C29EB473F'         // ciphertext
]
```

List. 7.7: OSCORE Request to the protected *temperature* resource

As discussed in Section 5.2.2.2, an OSCORE message is a `COSE Encrypt0` structure. The `unprotected_header` map contains the OSCORE sender ID and the sender sequence number. Since this request is a `GET` request, the ciphertext does not contain any data and thus only consists of the authentication tag produced by the authenticated encryption algorithm.

The OSCORE response to the above request is depicted in Listing 7.8.

```
[
  h'',                                // protected header
  {                                   // unprotected header
    / piv / 6: h'00',
    / kid / 4: h'42C7'
  },
  h'BB310AD937E37D9A8FEAA4B6510D32D9  // ciphertext
  FB581E64851563A778'
]
```

List. 7.8: OSCORE Response to the protected *temperature* resource

We can see how the `kid` parameter in the `unprotected_header` map is set to the resource server's sender ID and the resource server's sender sequence number encoded in the `piv` parameter is also set to zero. The decrypted payload in the ciphertext contains the protected resource `{ "temperature": "23C" }`.

7.2 Message Size

As mentioned in Section 4.3, we use HTTP over TCP to transmit messages between ACE entities. Since the ACE framework recommends implementations to use the CoAP protocol over UDP to further reduce the size of messages being transmitted over the medium, we have recorded the traffic generated by our implementation using the *Wireshark*[1] protocol analyzer with the intent to assess the overhead associated with the HTTP protocol with respect to the transported CBOR payload. To that end, we measured the size of

[1]https://www.wireshark.org

the messages exchanged during the ACE protocol flow. The results are summarized in Table 7.1.

Message	Payload	HTTP Packet	Overhead	Percentage [%]
Token Request	183	355	172	48.5
Token Response	385	536	151	28.2
Token Upload	270	447	177	39.6
EDHOC Msg 1	91	274	183	66.8
EDHOC Msg 2	229	385	156	40.5
EDHOC Msg 3	129	313	184	58.8
Temperature Access	20	196	176	89.8
Response	38	188	150	79.8
Update LED value	32	201	169	84.1
Response	23	178	155	87.1
Total	1400	3073	1673	54.4

Tab. 7.1: Recorded message sizes in Bytes for the message payload and whole HTTP packet

For every message exchanged during the course of the ACE protocol flow, we have measured the size of the CBOR payload as well as the size of the HTTP packet that is used to deliver the CBOR payload. Our results show that around half of the bytes exchanged between ACE entities can be attributed to HTTP. Most of this overhead is caused by the text-based HTTP headers, which means there is a lot of opportunity to further reduce the message size by using CoAP, a binary application layer protocol.

Furthermore, our results also show that the messages exchanged during the protocol flow are generally very concise, which we attribute to the use of the CBOR encoding format. To investigate the impact of CBOR, we compare some messages encoded as CBOR with a similar JSON formatted message in the following section.

7.3 Comparing CBOR and JSON

To assess the impact of using CBOR on the size of the ACE messages, we analyze the length of the encoded byte string produced by using CBOR as well as using JSON for the messages exchanged as part of the initial Token Request from the client to the authorization server as well as the access token itself.

Token Request From Listing 7.1 as well as Table 7.1 we can gather that the CBOR encoded payload for the Token Request from the client to the authorization server results in a payload length of 183 bytes. From these 183 bytes, 90 bytes can be attributed to the encoded COSE key structure in Listing 7.3. A JSON equivalent request body is illustrated in Listing 7.9, where the proof-of-possession key is encoded as defined in the JSON Web Key (JWK) standard in [19].

```
1 {
```

```
  "client_id": "ace_client_1",
  "client_secret: "ace_client_1_secret_123456",
  "grant_type": "client_credentials",
  "aud": "tempSensor0",
  "scopes": "read_temperature,post_led",
  "cnf" : {
    jwk: {
      "kty": "EC",
      "crv": "P-256",
      "kid": "my_key",
      "x": "1oSTcY2vQbPpLR0iPs2S6tZ0plwQSuAUfp8towGAOt8",
      "y": "V_ZUtQPvV6HP6wPbAqaQjdhRmuVxL3ucA6pWEgaYUMk",
      "alg": "ES256"
    }
  }
}
```

List. 7.9: Token Request parameters encoded as JSON

This JSON encoded version of the request parameters requires 419 bytes to be encoded. With only 183 bytes, the CBOR encoded parameters are 56.3% smaller.

Access Token Since the access token needs to be stored by the resource server, the size of the access token is of great importance for devices which are limited with respect to memory capacity. The use of small access tokens results in the resource server being able to store more access tokens without the client having to re-upload the access token. As suggested by the ACE framework, the tokens issued by our authorization server implementation use a CBOR Web Token to encode the authorization claims, which is illustrated in Listing 7.5. In conventional OAuth 2.0 using the JSON format, many implementations chose to use JSON Web Tokens whose format is defined in [20]. We illustrate a JWT that contains the same authorization claims as the CWT used in our implementation in Listing 7.10.

```
{
  "alg": "ES256",
  "typ": "JWT"
}
{
  "iss": "ace.as-server.com",
  "iat": 1533296406,
  "exp": 1533303606,
  "cti": "1710",
  "aud": "tempSensor0",
  "scopes": "read_temperature,post_led",
  "cnf": {
     "jwk": {
       "kty": "EC",
       "crv": "P-256",
       "kid": "my_key",
       "x": "1oSTcY2vQbPpLR0iPs2S6tZ0plwQSuAUfp8towGAOt8",
       "y": "V_ZUtQPvV6HP6wPbAqaQjdhRmuVxL3ucA6pWEgaYUMk",
       "alg": "ES256"
    }
  }
```

```
22 }
```

List. 7.10: JSON Web Token with equivalent claim set compared to a CWT

Signed with an elliptic curve key using the NIST P-256 curve, encoding this signed JWT produces a byte string of 542 bytes. In contrast, the CWT listed in Listing 7.5 only requires 270 bytes, resulting in a reduction of 50.2%. This result suggests that resource servers can store twice as many access tokens if the access tokens are encoded using CBOR instead of JSON.

7.4 Timing and Performance Measurements

To get an estimate for the performance that we can expect from our embedded resource server implementation, we have measured the time it takes to process some messages exchanged during the ACE protocol flow. We have summarized the obtained timing measurements in Table 7.2.

Context	Duration [ms]	Action	Duration [ms]
Token Upload	240	Verify CWT Signature	233
EHOC MSG 1	547	Generate ECDH Key Compute Shared Secret Sign MSG 2 Encrypt MSG 2	128 74 183 1
EDHOC MSG 3	342	Decrypt MSG 3 Verify MSG 3 Signature	3 172
Setup OSCORE Context	205	—	—

Tab. 7.2: Measured duration in milliseconds (ms) of certain actions performed by the embedded resource server.

We should note that the durations measured for the actions performed in a specific context do not necessarily sum up to the duration measured for the whole context. The actions only cover cryptographic computations whereas the durations for the whole context also includes time spent encoding and decoding CBOR objects or memory management.

We can see how most of the time is spent to setup the shared secret within the context of the EDHOC key exchange protocol. This is expected since this part of the message exchange requires a lot of asymmetric cryptography computations to be performed. Even with the hardware acceleration provided by the secure element, these operations can take around 200 milliseconds to complete.

In contrast, encrypting and decrypting messages using AES is very fast. We measured around one millisecond for encryption and three milliseconds for decryption. While we expected symmetric encryption as provided by AES to be fast, these measured durations are exceedingly short. This can be explained by the fact that the ESP32 microprocessor our embedded implementation is running on has special cryptographic hardware to accelerate AES encryption and decryption.

8 Outlook

In this chapter we present some usage examples that show how our implemented solution could be used in a production environment as well as some opportunities for future work.

8.1 Possible Implementations and Deployment Example

As of the time of writing this report, our implementation of the entities encompassed by the ACE framework primarily exists as a collection of software libraries. We currently do not offer an end-to-end solution that can be deployed by an end user in a simple manner.

In order to provide such a solution, we would have to implement a comprehensive provisioning software to replace the large amount of manual provisioning required by the current implementation. This provisioning software would provide means to register resource servers through an easy to use user interface and would configure the resource server with the required cryptographic properties. As such, the provisioning tool would pre-establish trust between the configured resource server and authorization server by exchanging their respective public keys. Such a software could be realized with the libraries developed in the context of this project.

Given the existence of a comprehensive provisioning software, we can imagine our implementation being used wherever there is a need for granular and controlled means of authorization for resources residing on constrained devices. In the following paragraph, we explore a usage example for the case of *home automation*.

Home Automation Home automation involves the installation of sensors and actuators throughout the living area. The sensors measure temperature, humidity, air quality or other values related to air quality or security in different parts of the living space. These sensors are attached to networking capable devices which inspect the values measured by the attached sensors and expose an endpoint for clients to access these values. Thus, these devices act as ACE resource servers. Similar to sensors, resource servers could expose endpoints to control actuators. For example, there could be actuators to control the lighting in different rooms or to raise and lower the window blinds.

All these sensor values and actuators are made accessible through multiple clients. One client could be a web server hosting a web page where all the sensor and actuator values can be visualized and changed. Other clients could be running on a user's smartphone or tablet. Only registered clients are authorized to access the resources associated with the sensors and actuators in the network. The security offered by the ACE profile would prevent unauthorized access to these protected resources even if an attacker is physically connected to the same network.

8.2 Future Work

On the basis of the work established as part of this project, there is a lot of opportunity to extend and improve on our findings.

Discovery In our implementation, the client implicitly knows how to access a protected resource on a certain resource server. In a real deployment scenario, the client would need a way to discover the protected resources available in the network. Additionally, the client also would need to discover the authorization server in charge of issuing an access token for a particular protected resource. Shelby et. al. propose the use of a *Resource Directory* in [21]. This resource directory can be queried by clients to retrieve the available resources along with their descriptions.

Grant Types As the ACE framework continues to be enhanced, more grant types may be integrated into the authorization flow. Our current implementation only supports the *Client Credentials* grant type, which would need to be adapted for the new ACE grant types.

Reference Tokens Self-contained access tokens have the advantage of not requiring any interaction with the authorization server when the resource server inspects the authorization claims contained in the access token. However, if the resource server is extremely limited in terms of memory capacity up to a point where it is no longer feasible to store a self-contained access token, a referential token could pose a solution. While our implementation of the authorization server is prepared to introspect reference tokens, it is currently impossible to issue reference tokens. This functionality would need to be added if referential tokens are required by the deployment scenario.

Application Layer Protocol As discussed in Section 4.3, we use HTTP as an application layer protocol even though the ACE framework recommends the use of the Constrained Application Protocol (CoAP). We have shown in our results in Section 7.2 that HTTP is responsible for about half of the bytes transmitted over the network. Given that CoAP usually uses UDP as its transport layer protocol, we expect that using CoAP instead of HTTP would help to further reduce the traffic generated by the ACE protocol flow and in turn result in longer deployment times for devices which draw their power from a battery. As soon as networking libraries offer comprehensive support for the mechanisms required by CoAP, we could adapt our implementation to use CoAP instead of

HTTP. Since our implemented security profile (EDHOC) is implemented on the application layer, we expect switching the protocol from HTTP to CoAP to be straightforward.

DTLS Profile As of writing this report, our implementation offers a single security profile to be used between the client and resource server, i.e. OSCORE via EDHOC. Adding support for the DTLS security profile as proposed in [17] could enhance the flexibility of our implementation.

EDHOC In the currently implemented version of the EDHOC key exchange protocol, we omit certain parameters to be sent in the EDHOC messages since all ACE entities already share these parameters implicitly, such as which elliptic curve to use or the type of encryption algorithm used. To implement the protocol to specification, our implementation needs to be extended with these parameters.

OSCORE In our implemented adaption of the OSCORE security profile for HTTP, we only encrypt the HTTP payload instead of the whole request including sensitive header parameters. As an improvement, the OSCORE context could also be used to encrypt additional request parameters such as HTTP headers.

Embedded Client In the scope of this project, we have developed a resource server implementation that is capable of running on a constrained embedded device. However, the authorization server and client are still only available as Python implementations. As a possible future work, our implementation could be enhanced with an embedded implementation for the client as well. This implementation could leverage some of the work done as part of the embedded resource server implementation.

9 Conclusion

In this project, we have implemented the entities and protocols proposed in a working document drafted by members of the Internet Engineering Task Force (IETF). The draft proposes a framework for controlled authorization and authentication for devices with limited processing and memory capabilities by extending the OAuth 2.0 standard. We have developed software that implements the three proposed entities, i.e. the authorization server, resource server and client and have demonstrated how the implemented entities communicate with each other in order to achieve an authorized access to a protected resource. In order to achieve secure communication between the client and resource server, we have implemented an application layer security protocol which encrypts the messages being exchanged between the client and resource server. Additionally, we have demonstrated a resource server implementation capable of running on constrained devices and enhanced the security capabilities by integrating a secure element with the embedded resource server. This secure element is used to increase the performance of asymmetric cryptography computations. Our results show that by using the binary message encoding format CBOR, we can achieve very concise messages exchanged between entities. We also document how HTTP is responsible for a large part of the generated message size leaving a lot of room for further improvements by using a binary application layer protocol such as CoAP.

References

[1] Ludwig Seitz, Goeran Selander, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. Authentication and authorization for constrained environments (ace) using the oauth 2.0 framework (ace-oauth). Internet-Draft draft-ietf-ace-oauth-authz-12, IETF Secretariat, May 2018. `http://www.ietf.org/internet-drafts/draft-ietf-ace-oauth-authz-12.txt`. 4, 18, 29

[2] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. `http://www.rfc-editor.org/rfc/rfc6749.txt`. 12

[3] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`. 6

[4] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. 34

[5] Hanqing Wu and Liz Zhao. *Web Security: A WhiteHat Perspective*. Auerbach Publications, Boston, MA, USA, 2015. 34

[6] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, RFC Editor, May 2014. `http://www.rfc-editor.org/rfc/rfc7228.txt`. 51

[7] J. Schaad. Cbor object signing and encryption (cose). RFC 8152, RFC Editor, July 2017. 21

[8] C. Bormann and P. Hoffman. Concise binary object representation (cbor). RFC 7049, RFC Editor, October 2013. 21

[9] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig. Cbor web token (cwt). RFC 8392, RFC Editor, May 2018. 23

[10] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[11] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *Proceedings of the The International Conference on Smart Card Research and Applications*, CARDIS '98, pages 277–284, Berlin, Heidelberg, 2000. Springer-Verlag. 8

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

[13] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7252.txt`. 6

[14] Goeran Selander, John Mattsson, and Francesca Palombini. Ephemeral diffie-hellman over cose (edhoc). Internet-Draft draft-selander-ace-cose-ecdhe-08, IETF Secretariat, March 2018. `http://www.ietf.org/internet-drafts/draft-selander-ace-cose-ecdhe-08.txt`. 40, 42, 43, 68

[15] Goeran Selander, John Mattsson, Francesca Palombini, and Ludwig Seitz. Object security for constrained restful environments (oscore). Internet-Draft draft-ietf-core-object-security-12, IETF Secretariat, March 2018. `http://www.ietf.org/internet-drafts/draft-ietf-core-object-security-12.txt`. 37, 38, 39

[16] Ludwig Seitz, Francesca Palombini, Martin Gunnarsson, and Goeran Selander. Oscore profile of the authentication and authorization for constrained environments framework. Internet-Draft draft-ietf-ace-oscore-profile-02, IETF Secretariat, June 2018. `http://www.ietf.org/internet-drafts/draft-ietf-ace-oscore-profile-02.txt`. 32

[17] Stefanie Gerdes, Olaf Bergmann, Carsten Bormann, Goeran Selander, and Ludwig Seitz. Datagram transport layer security (dtls) profile for authentication and authorization for constrained environments (ace). Internet-Draft draft-ietf-ace-dtls-authorize-03, IETF Secretariat, March 2018. `http://www.ietf.org/internet-drafts/draft-ietf-ace-dtls-authorize-03.txt`. 32, 64

[18] National Institute of Standards and Technology. *FIPS 186-4: Digital Signature Standard (DSS)*. July 2013. 55

[19] M. Jones. Json web key (jwk). RFC 7517, RFC Editor, May 2015. 59

[20] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. `http://www.rfc-editor.org/rfc/rfc7519.txt`. 14, 60

[21] Zach Shelby, Michael Koster, Carsten Bormann, Peter Van der Stok, and Christian Amsuess. Core resource directory. Internet-Draft draft-ietf-core-resource-directory-13, IETF Secretariat, March 2018. `http://www.ietf.org/internet-drafts/draft-ietf-core-resource-directory-13.txt`. 63

[22] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009. 7, 8, 9, 10, 11, 34, 40, 41, 68

List of Figures

List of Tables

Listings