Decentralized trust models for the Internet of Things

Master Thesis

Flurin Trübner

August 2020

Thesis supervisors:

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

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

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

Software
Engineering Group

Acknowledgment

I would like to thank my supervisors Prof. Dr. Jacques Pasquier-Rocha and Arnaud Durand for the possibility to work on the subject of this thesis and for their help.
Special thanks go to Arnaud Durand for always having an open door answering my questions and for the numerous discussions.

Abstract

This work introduces a new approach for authorization in the Internet of Things (IoT). An assessment of this approach is done by a proof of concept that builds on top of the Authentication and Authorization for Constrained Environments framework (ACE-OAuth). ACE-OAuth combines building blocks including OAuth 2.0 and the Constrained Application Protocol (CoAP) meeting specific requirements for the IoT under resource constraints.

This approach is extended by a decentralized trust model securing and monitoring the exchange of messages and authorization between participating nodes of the ACE-OAuth ecosystem. The decentralized trust model is based on pretty good privacy using a smart contract that runs on the Ethereum blockchain. This decentralization addresses problems resulting in the centralization of trust by third parties.

An implementation of a CoAP *Authorization Server* based on the ACE-OAuth specification is communicating with the decentralized trust model. This *Authorization Server* issues access tokens related to pre-established authorization on the smart contract. ACE-OAuth *Clients* request these access tokens to establish secured communication and to authenticate to *Resource Servers* in order to request a protected resource.

Finally, the feasibility of the introduced approach is assessed based on the costs that result from the usage of the decentralized trust model on the Ethereum blockchain.

Keywords: IoT, ACE-OAuth, Authentication, Authorization, Decentralized Trust Model, Web of Trust, Pretty good privacy, Smart Contract, Ethereum, Blockchain

Table of Contents

1. Introduction **2**

1.1. Motivation . . . 2

1.2. Authorization in the Internet of Things . . . 2

1.3. Goal . . . 4

1.4. Contribution . . . 4

2. Theoretical Background **5**

2.1. Introduction . . . 5

2.2. Authorization Framework: OAuth2 . . . 6

2.2.1. Roles . . . 6

2.2.2. Access Token . . . 7

2.2.3. Authentication Flow . . . 8

2.3. Constraints in the IoT . . . 9

2.4. Constrained Protocols . . . 10

2.4.1. Constrained Application Protocol (CoAP) . . . 10

2.4.2. Concise Binary Object Representation (CBOR) . . . 14

2.4.3. CBOR Object Signing and Encryption (COSE) . . . 17

2.4.4. Object Security for Constrained RESTful Environments (OSCORE) 20

2.4.5. CBOR Web Token (CWT) . . . 25

2.5. Authorization Framework: ACE-OAuth . . . 28

2.5.1. Building Blocks . . . 28

2.5.2. Roles . . . 29

2.5.3. Extensions for Constrained Environments . . . 31

2.5.4. Elliptic Curve Cryptography (ECC) . . . 31

2.6. Decentralized Trust Model . . . 33

2.6.1. Implications of a decentralized Trust Model . . . 37

2.7. Smart Contracts . . . 37

2.7.1. Ethereum Gas Prices . . . 40

3. Implementation 41
3.1. Introduction 41
3.2. Workflow 42
3.3. Technologies 45
3.3.1. Node.js 45
3.3.2. Smart Contract 47
3.4. Implementation 47
3.4.1. Client 47
3.4.2. COSE Adapter 49
3.4.3. Authorization Server 50
3.4.4. Token Claim Key Translator 53
3.4.5. OSCORE Security Context Adapter 54
3.4.6. Smart Contract 55
3.4.7. Smart Contract API 58
3.4.8. Resource Server 59

4. Results 61
4.1. Workflow Results 61
4.2. Gas Prices 65

5. Outlook 68
5.1. ACE-OAuth Module 68
5.2. Further Scenarios 69
5.3. Smart Contract 69
5.4. Decentralized Identifiers 69

6. Conclusion 70

A. License of the Documentation 71

1
Introduction

1.1. Motivation

The Internet of Things (IoT) introduces a transformation of physical objects to connect them to the internet [26]. "IoT refers to the networked interconnection of everyday objects, which are often equipped with ubiquitous intelligence. IoT will increase the ubiquity of the Internet by integrating every object for interaction via embedded systems, which leads to a highly distributed network of devices communicating with human beings as well as other devices." [10]
This transformation is applied on objects in a growing number of domains including industrial and private purposes. Motivations behind interconnecting objects are very broad. Whereas some implementations have the purpose of saving lives, the purpose of other deployments is purely to entertain. In the last years the numbers of IoT devices connected to the Internet has grown vastly as a result of advertised advantages, the increasing popularity of the IoT and evolving technologies enabling new use cases [19].
But the increasing popularity and number of IoT devices not only introduces improvements. As IoT devices get more popular their widespread deployment emerges new challenges. Services provided over the Internet have to protect private data and be secured against any criminal intentions. IoT devices are no exception concerning data security and privacy. Many concerns about the privacy arise when having sensors measuring and collecting data of individuals.
Devices connected to the Internet and data stored or sent among devices still imply the same problems of security and privacy after decades of research in security and development of more powerful hardware capable of more complex cryptographic operations.
Furthermore, the introduction of devices built on computational far less powerful chips expand the existing problems making security a main concern in the IoT.

In this thesis a new approach is presented to allow a secure method for authorization on constrained IoT devices.

1.2. Authorization in the Internet of Things

Authorization is an approval that is granted to an entity in order to access a system [28]. In this thesis the main focus is laid on the approval of authorization among IoT devices

connected to the Internet to access resources and services of reachable devices.
These resources offered by IoT devices are of various intents and importance and therefore they are implicating different demands for the security of the related data. Unauthorized accesses to humidity data of plants on a balcony are less critical than attempts accessing smart door locks.
Having an increasing number of IoT devices in expanding domains scales to the number of potentially insecure endpoints. Especially sensors gathering data affecting the privacy or security of people and businesses implicate a need of protection from unauthorized accesses.
Previous research results and industry standards in the domain of authorization and authentication may be transferred to the IoT directly or with certain adaptions.
In principle, the authentication and authorization protocol defined and implemented in this thesis is based on the "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)" [17] that is actively elaborated during the time this thesis is written.
IoT devices in general are limited in their available resources. These constraints include compromises on the available computing power, the size of the memory or on the size of the battery. As a result of the limitations concerning the hardware of IoT devices, adaptions of current standards for authentication emerge.
In the ACE-OAuth framework, research results and specifications in the field of authentication are combined in order to introduce a protocol, allowing secure authorization among devices with constrained computational power.

Decentralized Trust Models

Having constrained resources implicates further limitations on trust models in order to secure authorization over the Internet. Considering the widely used Public Key Infrastructure (PKI), several restrictions hold back their adoption in the IoT. "The Internet of Things has been slow to adopt PKI due to reasons both economic and technical. Instead, embedded systems often rely on pre-shared keys, which become problematic when those systems are connected to the Internet and become globally addressable. The keys must be installed before deployment, and because centralized resources must share a key with each device in order to communicate, a single server compromise can put the entire network at risk." [16]
In order to reply to the limitations on the trust models introduced by constraint hardware and to address security concerns of centralized solutions, a decentralized approach for the implemented trust model is chosen. In a centralized trust model a central authority is verifying identifications of entities by their public keys. The replacement of the central authority with a distributed solution results in a decentralized trust model [2]. For the implementation of this decentralized approach, a smart contract that runs on an Ethereum blockchain is introduced.

In section 2.6, the implications of using a decentralized trust model are presented.

1.3. Goal

This thesis is following two main goals:

The first goal of this thesis is to explore a new approach to securely authenticate IoT devices among each other over the Internet. For this purpose, a proof of concept is implemented to demonstrate that the introduced approach is adequate. This approach for the authentication is based on a decentralized trust model used along the ACE-OAuth framework and its suggested protocols for IoT environments. The main part of the implementation concerns the interaction of an *Authorization Server* with the decentralized trust model to verify access of a requesting *Client*.

As a second goal, the implementation aims to make the technologies and specifications related to this thesis more accessible which would allow faster prototyping for various scenarios.

1.4. Contribution

The main contributions of this master thesis are:

- Development of a concept for IoT authentication using a decentralized trust model
- Development of a decentralized trust model running as a smart contract on an Ethereum blockchain
- A proof of concept implementation of the concept combining the required technologies and specifications
- An overview of the protocols and specifications that are used in this work and which are relevant in the IoT
- An assessment of the feasibility of a decentralized trust model running on a blockchain in relation to its costs

2 Theoretical Background

2.1. Introduction **5**
2.2. Authorization Framework: OAuth2 . . . . **6**
2.2.1. Roles 6
2.2.2. Access Token 7
2.2.3. Authentication Flow 8
2.3. Constraints in the IoT **9**
2.4. Constrained Protocols **10**
2.4.1. Constrained Application Protocol (CoAP) 10
2.4.2. Concise Binary Object Representation (CBOR) 14
2.4.3. CBOR Object Signing and Encryption (COSE) 17
2.4.4. Object Security for Constrained RESTful Environments (OSCORE) 20
2.4.5. CBOR Web Token (CWT) 25
2.5. Authorization Framework: ACE-OAuth **28**
2.5.1. Building Blocks 28
2.5.2. Roles 29
2.5.3. Extensions for Constrained Environments 31
2.5.4. Elliptic Curve Cryptography (ECC) 31
2.6. Decentralized Trust Model **33**
2.6.1. Implications of a decentralized Trust Model 37
2.7. Smart Contracts **37**
2.7.1. Ethereum Gas Prices 40

2.1. Introduction

The connection of two distinct parts is most important for the approach in this thesis. On one hand there is the Authorization Framework that is concerning the necessary

messages to be sent in order gain authorization to a protected resource. On the other hand there is the approach to connect this framework with a decentralized trust model that is used to store the access information of the IoT devices.

Authorization Framework

As the implementation of this thesis is using the ACE-OAuth specifications that is based on OAuth2, the OAuth2 framework will be presented first. Then, the adaptions that the ACE-OAuth framework is introducing will be discussed.

Decentralized Trust Model

The interaction between IoT devices is complemented by a specific interaction between the entity that is issuing access and a decentralized trust model that is managing the authorization information. Therefore, the concept of decentralized trust models and their implications will be presented in this section. Further, smart contracts are introduced as a smart contract is used to model and deploy the decentralized trust model.
Additionally, the constraints existing on IoT devices will be presented in order to connect these constraints to the technologies that are suggested in the ACE-OAuth specification. The main goal of these suggested protocols is to reduce the requirements on computational power and power usage.

2.2. Authorization Framework: OAuth2

OAuth2 [5] is a standard for authorization flow that is widely used in the industry. The framework allows authorizing access on an owned resource that is provided by a different service in order to remove the need of exchanging user credentials. The representation of access is a token defining the granted access by the *Resource Owner*.

2.2.1. Roles

In the OAuth2 framework four roles are defined in order to authorize resource access without the exchange of user credentials beforehand:

Resource Owner
The *Resource Owner* is the entity that is owning a resource or a service which other entities want to access. In order to authorize access, the *Resource Owner* can grant access using the *Authorization Server* without the need of sharing its credentials.

Resource Server
On the *Resource Server* there are protected resources of the *Resource Owner*. The role of the *Resource Server* is to respond to resource requests from *Clients*. Resource Requests are accepted if the *Resource Server* receives a valid access token from the *Client* in its request.

Client
A *Client* is an entity that wants to access a protected resource belonging to a certain *Resource Owner*. The term *Client* has no implication on the implementation characteristics being an application executed on any device requesting a resource. In order to access a protected resource, the *Client* first has to retrieve the granted access from the *Resource Owner* as a token. The token then is sent in the *Client's* resource request to the *Resource Server*.

Authorization Server
The *Authorization Server* is the entity being responsible for issuing access tokens. Therefore, the *Authorization Server* first has to authenticate the *Client* successfully to obtain the corresponding access granted by the *Resource Owner*.

The same roles are present in the ACE-OAuth framework.

2.2.2. Access Token

In the authentication model of a traditional Client-Server architecture, a *Client* wants to access a protected resource on a *Server*. But in order to ensure that not only the owner of the resource has access to the resource in this architecture, the *Resource Owner* has to share its credentials with the *Client*. In that way the *Client* can access the protected resource using these credentials.
Sharing user credentials among multiple entities emerges security problems but also limitations in the authorization for accessing protected resources.
Not only third party applications representing a *Client* have to store the *Resource Owner's* credentials, but also the concerning *Resource Servers* have to support the exchanged credentials for authentication. If the *Resource Owner* has to share its credentials this implicates that at least two further entities have to store and transmit the credentials which introduces additional points of failure.
Another emerging problem by sharing credentials with a third party application is its unrestricted access. The credentials only allow one single set of access information as a scope or an expiry. This directly increases the complexity for the *Resource Owner* to share restricted access with third parties.
Another problem is the revocation of certain credentials. Sharing the same credentials with multiple third parties implicates that the revocation of these credentials results in the revocation of the access for every application the credentials have been shared with.

Tight coupling between the *Client*, *Resource Owner* and the *Resource Server* is the main problem for these issues. OAuth2 introduces an additional authorization layer in order to address these problems. An access token is the representation of access granted from a *Resource Owner* to a *Client*. By introducing tokens, there is no need for the *Resource Owner* to share its user credentials with another entity which is still common practice in many scenarios. "Instead of using the Resource Owner's credentials to access protected resources, the client obtains an access token – a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to third-party clients by an Authorization Server with the approval of the Resource Owner. The client uses the access token to access the protected resources hosted by the resource server." [5]

A token is similar to user credentials in terms of accessing a protected resource but a token may contain more information on the authorization.
As the token represents the granted access a *Client* received from a *Resource Owner* it is important for the *Resource Server* to be capable of processing these access tokens. The encoding of the access tokens as for the included authorization information may be different among implementations but plays a subordinate role for the *Client* itself. Different types of tokens exist. A token may simply contain security parameters as an identifier allowing the *Resource Server* to derive authentication using the parameters included in the token. Or the token may also contain directly some sort of authentication.

2.2.3. Authentication Flow

The OAuth2 protocol flow consists of four main messages that are sent between the *Client*, the *Authorization Server* and the *Resource Server*.

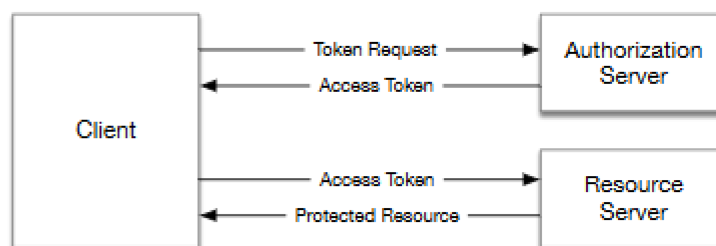


Fig. 2.1.: OAuth2 Authentication Flow, from [5]

Token Request
First, the *Client* sends a token request to an *Authorization Server* to gain access to a protected resource located on a specific *Resource Server*. Receiving the *Client's* token request, the *Resource Server* verifies that the *Resource Owner* has granted access for the requesting *Client* authorizing the *Client* to access the desired resource.

Token Response
If the validation of the authorization is successful, the *Authorization Server* generates an access token. This access token represents the access granted by the *Resource Owner* concerning the requested resource by the *Client* in the token request. After successful validation of authorization, the token is sent back to the *Client*.

Resource Request
In possession of the access token, the *Client* is authorized to access the protected resource on the *Resource Server*. Therefore, the *Client* includes the received access token in its subsequent requests to the resource endpoint.

Resource Response
As a last step needed to authorize the *Client*, the *Resource Server* has to validate the access token received in the *Client's* resource request. After successful validation of the received token, the request for the protected resource can be granted for this *Client*. If the

authorization was successful, the *Client* receives the protected resource from the *Resource Server* as response to its request.

2.3. Constraints in the IoT

As mentioned above, the constraints on the devices are a main challenge in the IoT. If it wasn't for those constraints, the current industry standards for authentication and authorization could simply be applied in the IoT. But these emerging challenges on the IoT devices are leading to new approaches and suggestions in order to maintain high security on the message exchange of data over the Internet.
These constraints are listed in this section to further connect them to the adaptions in the ACE-OAuth framework and the suggested protocols.

Low Current Capacity
In some deployments IoT devices cannot have a power connection but they have to be powered by batteries. This implicates limited energy availability until the battery is drained. Sending and receiving messages using wireless technology is causing a large amount of energy consumption in IoT devices.
Several optimizations result in a reduction of the power that required for sending and receiving messages.
A major part of the total amount of drained power is resulting from the number of messages as well as from the size of these messages. Therefore, it is important not only to reduce the number of messages but also the size of these messages.

Low Processing Power
IoT devices are often equipped with processors that have a significantly lower performance than processors used in other devices that are connected to the Internet, such as a smartphone. Especially in cryptographic operations this can lead to a delay in the protocols used. Therefore, protocols offering simple and computational inexpensive processing algorithms are preferred.

Low Amount of Memory
IoT boards are often equipped with a low amount of memory compared to other devices and therefore efficient algorithms and smaller implementations of protocols become more important. This further implies required adjustments in the cryptographic protocols. For example, a reduction of the size of the cryptographic keys or a protocol that is requiring less memory for the derivation and usage of the keys.

User Interfaces
Often the access to protected resources is given by a certain user interface. For example, in order to access an email mailbox one is prompted with input fields for the email address as well as the corresponding password. In IoT deployment scenarios it is often not possible or by design it might not make sense to offer a user interface.
These scenarios then better are delegated by using user-controlled devices, such as smartphones or tablets, which then communicate with the IoT device and authenticate the user using appropriate protocols.

Availability
Communication between IoT devices is not always possible and the availability of the devices can be very low in certain scenarios. Often the devices switch to sleep mode in order to save energy or networks might not be available consistently.

2.4. Constrained Protocols

In this section, the protocols which are suggested in the ACE-OAuth Framework in order to meet the needs in regard to the limited resources of IoT devices are presented. All of the mentioned protocols are used in the implementation done for this thesis.

2.4.1. Constrained Application Protocol (CoAP)

CoAP [32] is a transfer protocol designed for machine-to-machine applications and for the usage on constrained nodes and networks as often found in IoT scenarios. The main goal of CoAP is to optimize the Representational State Transfer (REST) and HTTP for machine-to-machine applications. As the CoAP protocol is based on the HTTP protocol it holds many similarities.

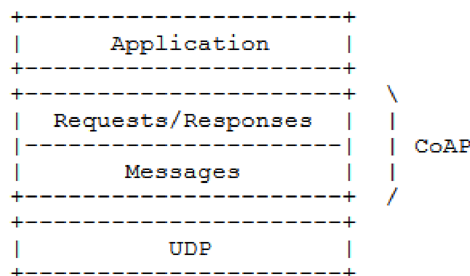


Fig. 2.2.: CoAP Layers, from [32]

Basically, a *Client* is sending a request message to a server and eventually receives a response. In CoAP and in machine-to-machine applications in general, the clear separation of *Client* and *Server* often disappears and a node resumes both roles: the role of the *Client* and of the *Server*.

Messaging Model

Message Types
The User Datagram Protocol (UDP) is used in CoAP to exchange messages among the endpoints. In order to achieve reliability using the UDP protocol, CoAP introduces four different message types.

Confirmable Message
Confirmable Messages require an acknowledgment from the receiver of the message. If no messages are lost, each confirmable message results exactly in one response message either of the type "acknowledgment" or "reset".

Non-Confirmable Message
Non-Confirmable messages do not require an acknowledgment by the receiver of the message. This might be used for periodic messages whose infrequent absence would not imply any issues.

Acknowledgment Message
An Acknowledgment Message is the response to a message of type "confirmable" and is stating that the message has been received.

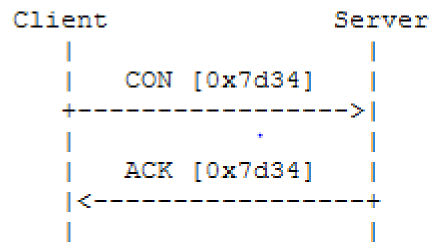


Fig. 2.3.: Reliable Messaging, from [32]

Reset Message
A Reset Message is sent as a response if the recipient of a message ("confirmable" or "non-confirmable") is lacking information in order to successfully process the received message. This might happen after messages that include necessary information were not received.

Message Format

One of the main motivations behind the CoAP protocol is to exchange messages over UDP having each CoAP message using one UDP datagram. This allows smaller message sizes and therefore reducing the required amount of messages. Additionally, no TCP connection has to be established and maintained.

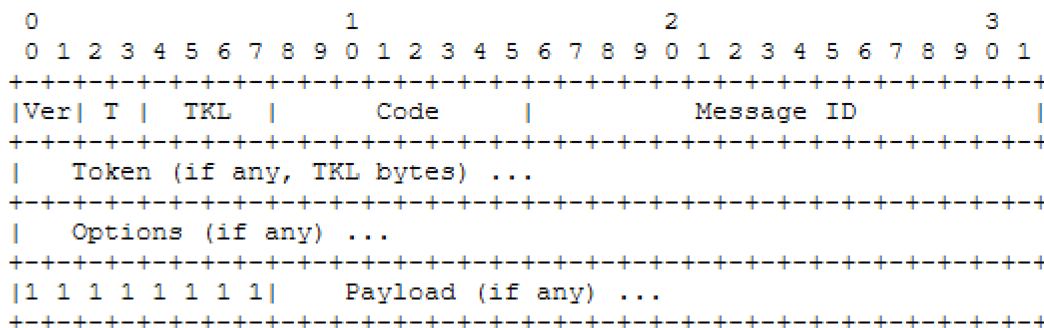


Fig. 2.4.: CoAP Message Format, from [32]

The message format specified for CoAP includes a header of a fixed size of four bytes. In the header the type of the message as well as further information that are needed in

order to process the message are stored.
The four bytes used by the header are followed directly by the data contained in the message. The data to be transferred contained in the message is following right after the four bytes used by the header.

Message Header

Version (Ver)
The Version is an unsigned integer of two bits indicating the version of CoAP that was used in order to generate the message.

Type (T)
The Type bits define the CoAP message type used for the message: "confirmable", "non-confirmable", "acknowledgement" or "reset".

Token Length (TKL)
The Token Length defines the variable length of the token that might be included in the message using four bits.

Code
The byte used to define the Code is split in two parts. The first three bits classify the code class "c" and the following five bits stand for the details "dd". This resulting in the "c.dd" format for the Code byte. For example, "0.02" defines a POST request.
The following classifications are defined in the CoAP specification:

- The Empty message is a special case and is specified as 0.00.
- Request messages are defined in the range of 0.01-0.31. Currently four methods are specified while the other values are unassigned.

```
+------+--------+
| Code | Name   |
+------+--------+
| 0.01 | GET    |
| 0.02 | POST   |
| 0.03 | PUT    |
| 0.04 | DELETE |
+------+--------+
```

Fig. 2.5.: CoAP Request Messages, from [32]

- Response messages are defined in the range of 2.00-5.31 and have more details specified than the Request messages.
- All other possible codes are Reserved Messages not being specified in CoAP.

Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable
4.12	Precondition Failed
4.13	Request Entity Too Large
4.15	Unsupported Content-Format
5.00	Internal Server Error
5.01	Not Implemented
5.02	Bad Gateway
5.03	Service Unavailable
5.04	Gateway Timeout
5.05	Proxying Not Supported

Fig. 2.6.: CoAP Response Messages, from [32]

Message ID
The two bytes that define the Message ID field in the header are used to match Acknowledgment and Reset messages to messages of the types Confirmable and Non-confirmable. Additionally, the Message ID is used for the detection of message duplicates.

Message Data
The actual data is appended in the message right after the message header.

Token
The Header is possibly followed by a token allowing of variable length between zero and eight bytes. The length of the actual token is defined in the Token Length Field of the header.

Options
The Options are used to exchange additional information. For example, to server a URI in a request or to support header metadata as the Content-Format used in HTTP.
If any Options are included in the message, the Options data is included after the token. Every instance of an Option is specifying its own Option Number that is based on a CoAP Option registry, as well as the size of the option value and the Option value itself.

Payload
If any Payload is included in the message, it is appended at the end of the message. The payload starts after a marker defined as a one byte field consisting of ones. The size of the payload is calculated by the size of the other fields as well as the size of the received UDP datagram.

CoAP URI Scheme

The CoAP and CoAP Secure URI scheme is similar to the URI scheme used for HTTP/S and is specified in the following format:

```
1 "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
```

List. 2.1: CoAP/S URI Scheme

2.4.2. Concise Binary Object Representation (CBOR)

CBOR [3] is a standardized binary representation of structured data following specific design goals not well addressed by current data formats. These design goals include a reduction in the message size as well as a reduction of the amount of the code that is required to process the messages.
CBOR is an extended version of the JSON data model but CBOR defines its own data model without planning on backwards compatibility.

Design Goals

Most common data formats used in Internet standards must be representable in the CBOR data format being unambiguously but it is not a requirement that data formats are uniquely encoded.

The amount of code required for the encoding and decoding has to be supported by devices which have limited memory and processing power as well as a reduced set of instructions. Therefore, a goal is a minimal amount of code using contemporary machine representations.

Data in CBOR format should be self-describing, allowing to decode the CBOR formatted data without the need of a schema description similar to JSON.

It is important to have a compact serialization in order to achieve small message sizes. But code compactness is rated more important than data compactness for CBOR. The size of JSON serializations is defined as an upper bound for the resulting CBOR data size, while the main goal is to achieve a lower implementation complexity.

Encoding and decoding must be frugal in CPU usage supporting both constrained devices and applications using a high volume of data.

The CBOR format must support all JSON data types in order to support conversions between JSON and CBOR.

The serialization used for CBOR must be backwards compatible to previous decoders while supporting extensibility. This is required to ensure a long lifetime of the CBOR format so that it can be used for decades. In order to support extensibility, it is important to provide a fallback for the decoders if the extensions are not understood.

CBOR Encoding

Each data item gets structured and encoded in the CBOR data format. All individual data items then get integrated in a further CBOR data type.
The initial byte of every data item specifies its major type and additional information. The major type is defined in the three high-order bits of this byte, representing one of the seven possible major types.

Additional Information
The additional information is specified in the five low-order bits, allowing values in the range of 0-31. Several usages are defined for the additional information depending on the resulting value:

- If the Value is less than 24, then the bits of the additional information are used as a small unsigned integer.
- A value in the range of 24-27 is directly followed by additional bytes to define a variable-length integer. A value of 24 is followed by one byte defining the length of the following value where a value of 27 is followed by an eight byte field used to define the value length. Values in-between specify two and four byte fields.
- Values between 28-30 are reserved for future extensions.
- A value of 31 specifies indefinite-length items.

The resulting integer value defined in the additional information is interpreted depending on the major type. In the case of integers, the resulting integer of the additional information is defining the integer value itself. For byte strings the additional information integer defines the length of the byte string data that follows the additional information byte.

Major Types
Seven major types are defined in the CBOR specification. The major type is defined with the first three bits of the initial byte of every data item. As stated above, the intention of the additional information is depending on the major type [3].

Major Type 0 (000): Unsigned Integer
0-23: The resulting integer of the additional information bits is the value itself.
24-27: Following integers of 1-, 2-, 4- or 8-Byte length define the size of the following data describing the value of the unsigned integer.
An Integer of 500 is encoded as 000-11001 followed by two bytes 0x01f4. 000 defines the major type 0, categorizing the data item as unsigned integer. The five bits 11001 result in the value 25 defining a two byte field that will follow. These two following bytes 0x01f4 then represent the unsigned integer of 500.

Major type 1 (001): Negative Integer
The encoding for negative integers is identical to the encoding of unsigned integers. Only the resulting value is interpreted as negative integer in comparison with the unsigned integer encoding.
The negative integer -500 is encoded as 001-11001 followed by the same two bytes 0x01f4 defining the value 500.

Major type 2 (010): Byte String
The additional information defines the following amount of bytes that is used to define the size of the following binary content that is representing the byte string. This representation is done analog to the encoding of the unsigned integer.
To define the length for a 500-bytes Byte String, the additional information would have to be 25, defining a two byte field that is needed to define the byte string size of 500. The two bytes 0x01f4 (500) following the initial byte 010-11001 define the length of the following byte string but not the value itself as for unsigned integers. These three bytes (010-11001 0x01f4) then are followed by 500 bytes of byte string data.

Major type 3 (011): Text String
The text string type is used in systems supporting human-readable text. Therefore, this major type allows to differentiate unstructured bytes and e.g. UTF-8 encoded text. The encoding is identical to the encoding of the byte string where the additional information defines the number of bytes that are used to specify the amount of following bytes used to describe the amount of text string data appended.

Major type 4 (100): Array of Data Items
This encoding is analog to the byte or text string encoding with the difference, that the following bytes that are defining the length of the string data are defining the number of data items that are included in the array. The bytes defining the major type and the amount of data are then followed by the data items.
The encoding for an Array containing 10 items would be 100-01010 where 01010 (10) defines the amount of following data items. If 500 data items would be included in the array, then the encoding would be 010-11001 0x01f4 followed by 500 data items.

Major type 5 (101): Map of Pairs of Data Items
A map is a pair of data items, where each pair consists of a data item as key and a data item as value. Maps in JSON are also called tables or objects. The encoding of a map is analog to encoding of an array where the additional information defines the number of the following pairs instead of the amount of data items for the array. By knowing about the map type the amount of data items can be derived from the amount of pairs.
The initial byte 101-01001 describes a map with 01001 (9) pairs and therefore 18 data items that are following the initial byte.

Major type 6 (110): Optional Semantic Tagging
This type is used to tag further types, as for example base64 strings.

Major type 7 (111): Data Types not needing content
Simple data types and floating-point numbers not needing content are specified as major type 7. Also the "break" stop code is included in this data type.

- 0-23: Simple value (0-23)
- 24: Simple value (32-255 in the following byte)
- 25-27: Floating point numbers
- 28-30: Unassigned

- 31: "break" stop code for indefinite-length items as well as for arrays and maps with indefinite length.

These eight different major data types lead to a jump table with 256 different possible semantics defined in the initial byte. CBOR decoders can be implemented based on a jump table [33] using the first byte and its 256 possible values. In constrained implementations the decoder can use the structure of the initial byte as described above for smaller code size.
Knowing these Major types, the array [1, [2, 3], [4, 5]] can be manually encoded to the CBOR format resulting in the serialization 0x8301820203820405 [3].

```
83 -- Array of length 3
     01 -- 1
     82 -- Array of length 2
        02 -- 2
        03 -- 3
     82 -- Array of length 2
        04 -- 4
        05 -- 5
```

List. 2.2: CBOR Array Encoding

CBOR Diagnostic notation

While the interchange of data is done in binary format, it is useful to have a human readable diagnostic notation. The diagnostic notation used in CBOR is based on the diagnostic notation of JSON. The notation used for the array in the encoding example above [1, [2, 3], [4, 5]] is exactly the diagnostic notation used in CBOR.

2.4.3. CBOR Object Signing and Encryption (COSE)

CBOR is defined as a small code and message size data format but there is a need for security services. The CBOR Object Signing and Encryption protocol [27] defines the creation and processing of signature, message authentication codes and encryption using CBOR, as well as the representation of cryptographic keys in CBOR serialization.

Basic COSE Structure

All COSE security messages are encoded as CBOR arrays.
The first three objects included in all COSE messages contain the same information. This is specified in order to reduce the amount of code needed to parse and process the different messages and to have common code to process a major amount of the different security messages. Elements following these three common objects depend on the type of the COSE message.

CBOR Tag	cose-type	Data Item	Semantics
98	cose-sign	COSE_Sign	COSE Signed Data Object
18	cose-sign1	COSE_Sign1	COSE Single Signer Data Object
96	cose-encrypt	COSE_Encrypt	COSE Encrypted Data Object
16	cose-encrypt0	COSE_Encrypt0	COSE Single Recipient Encrypted Data Object
97	cose-mac	COSE_Mac	COSE Mac-ed Data Object
17	cose-mac0	COSE_Mac0	COSE Mac w/o Recipients Object

Fig. 2.7.: COSE Message Types, from [27]

Protected Header Parameters
The protected header parameters are cryptographically protected and they are empty if protected parameters are not needed in the cryptographic computation. The protected header parameters are a protected map wrapped in a binary string object that is CBOR encoded.

Unprotected Header Parameters
Unprotected header parameters contain information about the security message not needing cryptographically protection. **Message Content**
The message content contains the plaintext or the ciphertext depending on the type of the security message.

Common Header Parameters

The COSE specification defines Common Header Parameters that are used to add information to the COSE messages.

alg
The alg parameter describes the algorithm that is used for security processing. This parameter must be authenticated by e.g. placing it in the protected header object if possible.

content type
The content type of the message content is defined in the content type parameter and might specify an integer or textual data among other types.

kid
The kid data item contains data that might be used as an input to derive the cryptographic keys intended by the message. The kid is not security critical and can be transmitted in the unprotected header object.

Initialization Vector (IV)
This parameter contains the initialization Vector that is the nonce for symmetric encryption algorithms that might be used to establish a security context to build a secured channel.

Partial IV
The Partial IV is part of the IV that is used to alter the value of the IV.

Signing

Two different signature messages are defined in the COSE specification. In the implementation for this thesis the COSE-Sign1 message is used. The COSE-Sign1 message is a single signer data object containing both the payload and the signature of the payload as well as meta information about both of them.
If the "ECDSA w/ SHA-256, Curve P-256" signature algorithm is used, the resulting array would result in the following structure in CBOR diagnostic notation.

```
[
    protected:
    {
        1(alg) :-7 (DSA 256)
    },
    unprotected:
    {
        4(kid):'11'
    },
    payload{},
    signature{}
]
```

List. 2.3: COSE-Sign1 Structure

The signature object is a CBOR map and may contain multiple signatures. Each of the signatures is defined as a COSE-signature object which consists of its own headers and a computed signature.

```
COSE-Signature
[
   Headers,
   signature : bstr
]
```

List. 2.4: COSE-Signature Object

To create a signature, a well-defined byte stream representation is needed. Therefore, the Sig-structure is introduced to create a canonical form. The Sig-Structure is used for the computation and verification of signatures in COSE.

```
Sig-structure
[
   context : "Signature" / "Signature1" / "CounterSignature",
```

```
4     body_protected : empty_or_serialized_map,
5     ? sign_protected : empty_or_serialized_map,
6     external_aad : bstr,
7     payload : bstr
8  ]
```

List. 2.5: COSE Sig Structure

Signature Computation
The following steps are required in order to compute the signature:

1. Creating a Sig-structure and populating the fields with the appropriate values.
2. Encoding of the created Sig-structure to a byte string that is defined as "ToBeSigned", the byte string that will get signed.
3. Passing the appropriate parameters to the signature creation algorithm. The key that is used to sign, the algorithm to be used for the signature creation and the "ToBeSigned" byte string created before are needed.
4. The signature returned by the signature creation algorithm then is placed in the signature field of the COSE-Signature object.

Signature Verification
The steps in the verification algorithm are identical to the steps required in order to compute the signature with a difference on the algorithm called. Instead of calling the signature creation algorithm now the signature verification algorithm is used. The verification algorithm takes same parameters but additionally also the signature to be verified.

Encryption

To support encryption there are two different encryption structures defined in the COSE specification. If the key that is used for the cryptographic operations is known implicitly, then the COSE-Encrypt0 structure is used. In all other cases the COSE-Encrypt structure is used.
The algorithms for encryption and decryption are similar to the algorithms in signing. First, an Enc-Structure is derived that will get encoded as a byte stream that is representing the Additional Authenticated Data (AAD). The AAD is used to derive an encryption key which is passed to an encryption algorithm together with the plain text to be encrypted.

2.4.4. Object Security for Constrained RESTful Environments (OSCORE)

The Object Security for Constrained RESTful Environments [12] (OSCORE) Protocol defines a method to extent CoAP by application-layer protection using COSE with Authenticated Encryption with Associated Data (AEAD) [23].

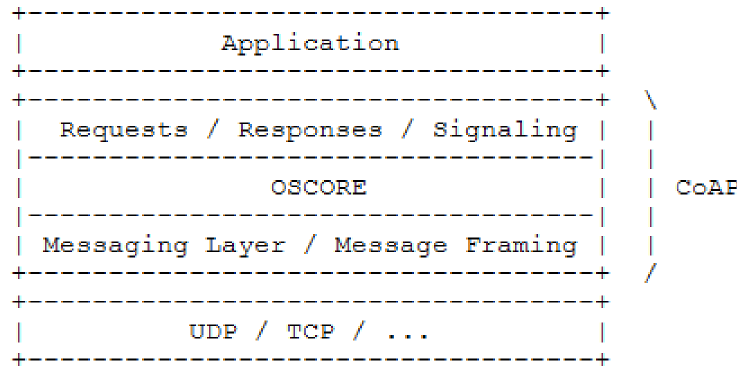


Fig. 2.8.: CoAP Layering using OSCORE, from [12]

OSCORE allows CoAP endpoints to establish an end-to-end protection. Therefore pre-shared keys may be used or keys may be established by using a key exchange protocol. Secure communication is then achieved by converting CoAP or HTTP messages to OSCORE messages. In this process HTTP messages first are converted into CoAP messages. In order to create an OSCORE message, the CoAP message is protected using COSE encryption. This encrypted message then includes in its header fields both the OSCORE options as well as the message fields of the COSE-Encrypt object.

In the implementation of this thesis, OSCORE is used to establish a security context that is used in the COSE encryption of the CoAP payload. Therefore, the Security context and its derivation is the main part of this section.

Security Context

To enable both the sender and the receiver to encrypt and decrypt messages sent, a security context has to be pre-established.
This security context corresponds to a set of parameters that are required for cryptographic operations in OSCORE. The security context consists of three sub contexts: the "common context", the "sender context" and the "recipient context".

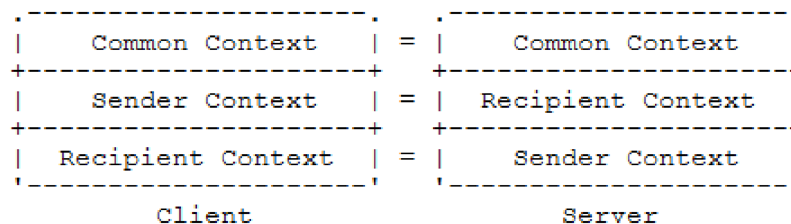


Fig. 2.9.: Matching of Contexts, from [12]

The common context is derived first and is then used together with additional data in order to derive the sender and recipient context. Two endpoints that want to communicate over an OSCORE secured channel derive each a sender and a receiver context. In order to receive asymmetric keys, the sender and receiver IDs are used in reverse so that the sender context of the first endpoint is matching the receiver context of the second endpoint and vice versa.

Common Context

In order to derive the sender and receiver contexts used for the cryptographic operations, the comment context has to be derived first on both endpoints and consists of the following parameters.

Common IV
The common IV is a byte string that is derived in order to generate an AEAD nonce. For the derivation of the common IV, the master secret, the master salt and the ID context are needed.

AEAD
The "Authenticated Encryption with Associated Data" algorithm will be used in the COSE encryption algorithm.

HKDF
HKDF is an identifier for HMAC-based key derivation functions. The function specified in the identifier is used to derive the sender and receiver keys, as well as the common IV.

Master Secret
The master secret is a random and variable length byte string that is required to derive the common IV and the AEAD keys. In the implementation of this thesis the master secret is derived by the Elliptic-Curve Diffie–Hellman (ECDH) protocol.

Master Salt
The master salt is an optional parameter that is used in the derivation of the AEAD keys and common IV.

ID Context
This parameter is providing additional information to identify specific contexts if needed. For example, if one sender ID is connected to multiple contexts the ID context is used to distinguish the different contexts related to the same sender ID.

Sender and Receiver Context

Both the sender and the receiver contexts consist of the same parameter. The only difference is the reversal of the keys in these contexts so that a sender key matches a recipient key.

Sender/Recipient ID
Both the IDs of the sender and receiver are used to identify an existing context. Additionally, both of the IDs are used to derive the common IV as well as the AEAD keys and for the derivation of the AEAD nonce if one is used.

Sender/Recipient Key
Both of the keys are derived from the Common Context together with the corresponding ID. These are the cryptographic keys used in the encryption and decryption and therefore in the establishment of a secure channel between two endpoints.

Sequence Number
The sequence number can be used for an enumeration of messages sent between endpoints. Further it may be used as an additional input for the derivation of unique AEAD nonces.

Replay window
This parameter might be used together with the sender sequence number. The replay window then is used for replay protection and to verify received requests.
In order to protect from replay of requests, an endpoint might verify, that the sender sequence number included in the COSE object was not received beforehand.

Security Context Derivation

In order to establish a secured channel using OSCORE, several input parameters are needed to first derive the security context required for encryption and decryption of the messages sent. Not all of these parameters are mandatory to derive the security context. The parameters that are not mandatory have default values to be used if none are transmitted.
The mandatory parameters for the derivation of the security context are both the sender and recipient IDs as well as the master secret.
The identifier of the AEAD/HKDF algorithms and the master salt are not mandatory. If no values are provided, the AEAD is using the AES-CCM-16-64-128 algorithm and the SHA-256 algorithm is used for the HKDF.

The mandatory and optional parameters have to be known and agreed on both endpoints deriving a security context to communicate over a secured channel. The strategy of the pre-establishing of these parameters may vary on the implementation.

Derivation of the Common IV and Cryptographic Keys

In order to receive the keys and the common IV, the HKDF algorithm that both endpoints have agreed on is used.

Derived Parameter = HKDF(master salt, master secret, info object).

The master salt and the master secret stay the same for all of the three parameters that get derived using the HKDF function. The parameters included in the info object define which parameter is being computed by the HKDF function.
The info object is a serialized CBOR array containing all parameters that are needed for the derivation except from the master salt and master secret.

```
1 info =
2 [
3     id : bstr,
4     id_context : bstr / nil,
5     alg_aead : int / tstr,
6     type : tstr,
```

```
7     L : uint,
8 ]
```

List. 2.6: Info Object Parameter

id
This field specifies the ID to be used for the derivation. The sender id is included to derive the sender key and the recipient id is used for the derivation of the recipient key. The field is empty for the derivation of the common IV.

id-context
If an ID context is needed it is transmitted in this field. The value is "nil" if no ID context is provided.

alg-aeag
The AEAG algorithm that is used during the HKDF call can be defined in this field. The default value for this field is 10 which is specifying the AES-CCM-16-64-128 algorithm, if no other specifier is provided.

type
This field is used to define which type of parameter will be derived from the HKDF function. The two possible values are "Key" to derive the sender/recipient key or "IV" to derive the common IV.

L
The length of the key or nonce that are used for the AEAD algorithm is defined in this field. The length is specified in the amount of bytes. A length of 16 bytes is defined for the derivation of the keys and a length of 13 bytes is defined for the derivation of the common IV.

The HKDF function consists of two steps. In the first step, the HKDF-Extract derives a fixed-length pseudo random key using its input. The HKDF-Expand is the second step, that expands the generated key of the first step into different key representations being the output of the function.

K = HKDF-Extract(Master Salt, Master Secret)
Output = HKDF-Expand(K, info)

If the length L defined in the info parameter is smaller than the output of the hash function, then the first L bytes of the expand output describe the derived parameters. These are the first 16 bytes of the output for the sender/recipient key and the first 13 bytes of the output for the common IV.

Example Derivation of a Security Context

In the following an example for the derivation of sender/recipient keys and a common IV on one endpoint is given. For the optional parameters the default values are used.

Input Parameters

- Master Secret: 0x0102030405060708090a0b0c0d0e0f10
- Sender ID: 0x00
- Recipient ID: 0x01
- Sender Key Info Object: [h”, null, 10, "Key", 16] (0x8540f60a634b657910)
- Recipient Key Info Object: [h’01’, null, 10, "Key", 16] (0x854101f60a634b657910)
- Common IV Info Object: [h”, null, 10, "IV", 13] (0x8540f60a6249560d)

Having created the info objects for the derivation of the keys and the common IV, the HKDF Function can be invoked passing the appropriate input parameters. As no algorithm is specified for the HKDF function, the HKDF SHA-256 is used as default.

Output Parameters
The same master salt and master secret are used to derive the sender/recipient keys and the common IV but the key info objects are used to derive the keys and vice versa for the common IV.

Output = HKDF(Master Salt, Master Secret, info)

As the sender/recipient keys are the first 16 bytes of the output and the common IV is defined by the first 13 bytes of the output the following output parameters are derived using the appropriate info objects.

- Sender Key: 0x321b26943253c7ffb6003b0b64d74041
- Recipient Key: 0x854101f60a634b657910
- Common IV: 0xbe35ae297d2dace910c52e99f9

The second endpoint would derive the security context identically. The only difference would be, that this endpoint would exchange both the sender and recipient IDs and therefore exchanging their info objects. As a result, the second endpoint would derive a recipient key that is identical to the sender key that was derived on the first endpoint and vice versa.

2.4.5. CBOR Web Token (CWT)

The CBOR Web token is derived from the JSON Web Token (JWT) using CBOR encoding. The CWT [21] specification is defining a compact solution to exchange representing claims between endpoints. Claims are key/value pairs representing information that is mostly about subjects in this context. As CWT is derived from JWT, the claims specified for JWT are used as well for CWT. As an addition to the CBOR encoding, the COSE signing and encryption protocols are used to provide application-layer security to the tokens.

CWT Claims

A CWT consists of a set of Claims. Each Claim consists of a Claim key, identifying the claim and the Claim Value representing the value of the claim. The Claim name is a human readable identifier. A set of registered claims are defined in the CWT specification.

Name	Key	Value Type
iss	1	text string
sub	2	text string
aud	3	text string
exp	4	integer or floating-point number
nbf	5	integer or floating-point number
iat	6	integer or floating-point number
cti	7	byte string

Fig. 2.10.: Defined Claims, from [21]

iss
The iss claim identifies the issuer of the token. In this thesis this is the *Authorization Server* of the ACE-OAuth framework.

sub
The identifier of the tokens subject is stored in the sub claim. This identifier specifies the principal that is requesting the token. In this context the ACE-OAuth *Client* is the requesting principal.

aud
The audience claim specifies the recipient that is intended for the token. The recipient must identify itself as a value in the audience claim or reject the token. The audience is the *Resource Server* concerning the ACE-OAuth framework.

exp
The Expiration Time claim defines the duration for the token to be valid. In order to be valid, the date value of the expiration time claim has to be in the future.

nbf
The "not before" claim is similar to the exp claim but defines a starting point for the token to be valid. The token must be rejected if the nbf claim value is not in the past.

iat
With the "issued at" claim the date on which the token was issued is specified. This claim allows determining the age of the token.

cti
The CWT ID Claim is used to transmit a unique identifier for this specific token.

Proof of Possession

A proof of possession (pop) token is a token that is bound to a cryptographic key. If a pop token is desired, the issuer of the token can include a cnf claim to the token. The issuer of the token declares with the cnf claim that the subject of the token is in possession of a particular key. This cnf claim can later be used by receiver of the pop token to cryptographically prove, that the sender of the token is in possession of this particular key. This proof of possession is defining the according token type.
The cnf claim is a CBOR map that is defining the proof of possession key [20].

COSE Key Claim
A CWT might contain a COSE-key object in its cnf claim to provide proof of possession for the token. In the implementation of this thesis an Elliptic Curve Digital Signature Algorithm is used to create COSE-sign objects that provide application layer security using asymmetric keys. To proof the possession of a private key the corresponding asymmetric key is included in the token as COSE-key object.
In addition to the public key, also the type of the key is included in the kty claim as well as the type of elliptic curve that is defined in the crv claim.

```
1 COSE-Key:
2 {
3     kty: EC,
4     crv: P-256,
5     x: a2a230b17ba9f8a93087da5486f531...,
6     y: 0b69de6d4bb1d06774ea83ed9de1...
7 }
```

List. 2.7: COSE-Key in a PoP Token cnf Claim

Example CWT

This is a possible CWT that is secured with COSE sign and encoded as a CBOR map. In order to reduce the size of the CWT the claim names are replaced by integers.

```
1  18
2  (
3     [
4        / protected /
5        {
6          / alg / 1: -7 / ECDSA 256 /
7        },
8        / unprotected /
9        {
10         / kid / 4: h'4173796d6d65747269634543445341
11                    23536' / 'AsymmetricECDSA256' /
12       },
13       / payload /
14       {
15         / iss / 1: "coap://as.example.com",
16         / sub / 2: "erikw",
17         / aud / 3: "coap://light.example.com",
```

```
18          / exp / 4: 1444064944,
19          / nbf / 5: 1443944944,
20          / iat / 6: 1443944944,
21          / cti / 7: h'0b71'
22        },
23        / signature / h'5427c1ff28d23fbad1f29c4c7c6a555e601d6fa29f
24                         9179bc3d7438bacaca5acd08c8d4d4f96131680c42
25                         9a01f85951ecee743a52b9b63632c57209120e1c9e
26                         30'
27    ]
28 )
```

List. 2.8: Example CWT Token

2.5. Authorization Framework: ACE-OAuth

The ACE-OAuth [17] Framework is derived from the OAuth2 framework and is specified for authentication and authorization in the Internet of Things addressing the constraints of many devices used in this quickly developing domain.

2.5.1. Building Blocks

To address the different constraints and needs in the IoT, the ACE-OAuth framework is based on four building blocks:

- The OAuth 2.0 framework is widely used in and supported by many IoT devices without any extensions needed. This makes it ideal to build on top of OAuth 2.0 but providing additional profiling for specific constrained settings.
- The ACE-OAuth framework builds on CoAP in deployments with communication environments, where a lightweight web transfer protocol is preferred over HTTP.
- As a third important building block, ACE-OAuth relies on CBOR for encoding wherever JSON is not compact enough.
- To support object- and application-level security, the ACE-OAuth framework is suggesting COSE and OSCORE.

By combining these four building blocks, the ACE-OAuth framework allows various deployments addressing specific needs in the domain of the IoT.

Design Decision

The ACE-OAuth specification defines several design decisions in order to address specific needs which are not well enough covered in the OAuth2 framework.

CBOR
It is recommended using CBOR as data format. If CoAP is used as web transfer protocol, then CBOR is required for the encoding.

COSE
If application layer security is needed for the CBOR encoded data, then the usage of COSE is recommended to provide security.

CWT
If self-contained tokens are required, then the specification is recommending the usage of CWT.

CoAP
The usage of CoAP instead of HTTP is recommended in the specification. But this is not precluding additional protocols addressing further constraints as with Low Energy Bluetooth.

Proof-of-Possession
The framework uses proof-of-possession tokens in order to increase the difficulty of token theft and therefore to improve the security.

Authz-Info endpoint
The framework introduces the authz-Info endpoint in order to reduce the message size and code complexity of requests received by the RS. In OAuth2 every request typically includes the access token. Instead of sending the token along in every request, the token is uploaded in a former and single POST message.

Client Credentials Grant
Having machine-to-machine communication and constraints on user interfaces, this framework recommends the usage of the client credential grant removing the need of manual actions by the *Resource Owner*. This results in the need of pre-established authorization from the *Resource Owner*.

Introspection
As the communication between the *Resource Server* and the *Authorization Server* is assumed, an introspection endpoint is recommended. The introspection endpoint allows the *Resource Server* to query the *Authorization Server* for claims associated with tokens and to further validate received tokens. This allows the *Client* to receive only a reference as a long-term token if the communication between the *Client* and the *Authorization Server* might not be ensured.

2.5.2. Roles

The ACE-OAuth framework is using the same roles as defined in the OAuth2 framework. Each of these roles has specific responsibilities in the ACE-OAuth framework.

Resource Owner

- Is responsible for registering the *Resource Server* on the *Authorization Server*
- Ensures the possibility for the *Client* to discover the *Authorization Server* that is issuing access for the *Resource Server*

- On client-credential grant, the *Resource Owner* is responsible for up-to-date access control policies on the *Authorization Server* concerning the *Resource Server*

Requesting Party

- Ensures that the *Client* possesses the credentials needed to authenticate on the *Authorization Server*
- Ensures the appropriate client configuration on the security requirements
- Registers the *Client* on the *Authorization Server* including the security configuration used by the *Client* if needed

Authorization Server

- Registers the *Resource Server* with their corresponding security context used to issue relating tokens
- Has *Clients* with their corresponding authentication credentials registered
- Gives the *Resource Owner* the possibility to configure and update access control related to the *Resource Server*
- Allows *Clients* to request tokens by exposing the token endpoint
- Authenticates token requests from *Clients*
- Processes token requests by rejecting or issuing tokens
- Processing introspection requests if an introspection endpoint is provided

Client

- Discovers the *Authorization Server* that is responsible for authorizing requests to the *Resource Server*
- Requests tokens by authenticating itself to the *Authorization Server*
- Processes received access tokens including the access information and the security parameters
- Ensures a safe storage of the proof-of-possession key
- Uploads the token to the *Resource Server* with a POST message to the authz-info endpoints on the *Resource Server*
- Requests resources on the RS after successful verification of the uploaded token
- Processes responds related to the resource requests sent to the *Resource Server*

Resource Server

- Ensures an endpoint to upload Access tokens. This is by default the authz-info endpoint
- Validates and stores access tokens received on the upload endpoint which includes the verification of the issuer and subject, the integrity of the token as well as the expiration and the storing of the token related to the matching request context
- Handles requests sent by the *Client* by arranging a secured context and authenticating the *Client* by auditing the stored tokens
- Responds to the *Client's* request according to the outcome of the token verification

- Ensures safe storage of security related credentials as proof-of-possession keys

2.5.3. Extensions for Constrained Environments

The ACE-OAuth framework recommends further extensions to the OAuth2 framework in order to address constraints in the IoT.

Credential Provisioning
In IoT deployments, it cannot be assumed that there is a Common Key Infrastructure that includes the *Client* and the *Resource Server* and therefore credential provisioning on the AS is used to allow authentication by binding these provisioned credentials to the access token.

Proof-of-Possession
By default, pop tokens are issued to enable the *Client* as the token holder to prove its possession of an access token and therefore its possession of the asymmetric public key that is bound to the token.

2.5.4. Elliptic Curve Cryptography (ECC)

ECC introduces advantages in performance at higher security levels including a protocol for a Diffie-Hellman key exchange protocol making use of elliptic curves [7].
This section is based on an Enuma Technology article [18].

Cryptographic Signatures

The intend of a cryptographic signature is to prove that one is in possession of a certain cryptographic key but also, that a message did not change since it was signed.

Protecting the Private Key

If someone obtains a copy of the private key used to sign, then this private key can be used to sign malicious messages. Therefore, it is important for the participants in the ACE-OAuth framework to ensure the security of the keys used in communication and authorization.
But storing keys securely is another constraint having limited resources. A common method is to regenerate the keys every time they are needed by using a key derivation function. Therefore, the keys don't have to be stored and cannot be copied. But requiring additional memory or cpu cycles contradicts to constraints being present in IoT deployments.
Another approach is to use hardware signing having which means that the private keys are stored in the hardware such that the keys cannot be accessed as it would be possible if they were stored in memory. But including such methods to increase the security increases simultaneously the requirements on the deployments as on hardware and pre-established information.

Elliptic Curves

Elliptic curves are mathematically defined by an equation that defines its points in a certain coordinate system.

$$y^2 = x^3 + a \cdot x + b$$

In the context of Elliptic Curve Cryptography, a private key is a random positive integer that is usually denoted by d.
The corresponding public key is denoted by Q and specifies a point on an elliptic curve that is defined by the equation above. The point therefore has two coordinates denoted by x and y. The public key Q is calculated by multiplying a point of the curve G (generator) and the private key d.

$$Q = d \cdot G$$

This equation can be reformulated to an equation having G added up d times. The result of the summation of points on the elliptic curve is another point on the same elliptic curve. Elliptic Curve Cryptography makes use of the fact that a point on the elliptic curve can be calculated easily but the parameters used to derive this point are computationally very expensive.

The Elliptic Curve Digital Signature Algorithm is used as well for COSE signing and requires the following steps to compute the signature:

- The signer is choosing a random number that is denoted by k.
- Using this chosen random number k, the signer then calculates a point on the curve that is denoted by C.
- C is calculated as described above by $C = k \cdot G$, where G is the Generator base point.
- The digest that is being used is denoted by e. It is calculated as the hash of the message to be signed.
- The signer now calculates a positive integer $s = (e + r \cdot d)/k$ using the random number k, the digest e and the private key d as well as the x coordinate of C denoted by r.
- Finally, the ECDSA signature is (r, s).

Having the signature (r, s) as well as the signers public key, it can be proven that the signer possesses the private key d that was used to derive the signature. Furthermore, the authenticity of the message can be proven, making use of the fact, that calculating a point on the curve by multiplication is simple.

2.6. Decentralized Trust Model

Introducing a decentralized approach addresses multiple challenges of centralized trust models in IoT deployments: On one hand, it addresses constraints on computational power of IoT devices. On the other hand, the introduction of a decentralized trust model deals with security concerns of centralized trust models such as single points of failure. Further, domains, email addresses and other digital identities are rented by using certificates, DNSs and further services that are offered by third parties. Therefore, centralized solutions result not only in security issues but also in usability challenges [4].
When those third party controlled systems were designed there were no solutions allowing a decentralized approach to agree on a certain state in a decentralized approach. Therefore, solutions relying on and trusting in third parties, which are managing identification, including identifiers and public keys, such as Public Key Infrastructures (PKI), were introduced and established.

In this section the security requirements and a threat model of IoT authentication are presented. Then, weaknesses of centralized solutions in the IoT are shown in order to present advantages of a decentralized trust model in IoT authentication.

Security Requirements in IoT Authentication

The following security goals ensure resilience and sustainability for authentication in IoT deployments [22]:

Integrity
Messages sent over the Internet must not be changed during their transmission and only authorized entities are allowed to modify stored data.

Availability
Participating entities must be able to verify other identities and authorizations at any time. Therefore, services offering corresponding data must be resilient, for example on denial of service (DoS) attacks.

Scalability
The amount of data and the amount of participants of a system mustn't have any impact on its performance.

Non Repudiation
Entities must not be able to deny performed actions as sending a message.

Identification
It must be possible to identify the participating entities.

Mutual Authentication
Participating entities must be able to authenticate each other by proving their identity in order to prevent spoofing of the entities identification.

Threat Model

In this section, a threat model [22] for IoT authentication is presented. The threat model includes a network model, an attacker model and possible attacks on IoT deployments which are described in the following:

Network Model

- Multiple nodes offer and use different centralized or decentralized IoT services.
- Every entity may communicate with numerous other participating entities.
- The network used to exchange messages is unreliable and potentially lossy.
- Not all participating entities can be trusted and the possibly high amount of participating entities is increasing the risk that entities are compromised.
- The network provides packet forwarding but it is not providing security guarantees as integrity or authentication.

Attacker Model

- Possible malicious users can access the network allowing them to sniff, drop, replay, reorder, inject, delay and modify messages.
- Attackers may participate in the ecosystem using more powerful machines and therefore, they are not impacted by the same hardware constraints as IoT devices.
- Participating nodes are assumed to be protected against physical attacks as accessing the device's memory or storage in order to retrieve secrets as private keys.

Possible Attacks

- **Sybil Attack** [8]: An attacker is adding multiple entities to the ecosystem in order to increase its possible influence on the system.
- **Spoofing** [25]: The identity of an entity is spoofed in order to gain its authorization.
- **Denial of Service** [24]: Attacks compromising the availability of a system by exploiting protocol flaws or by flooding it with a big amount of requests.
- **Message Replay** [29]: An attacker stores messages sent on the network and replays them later with a malicious intent.

PKI Weaknesses

Centralized solutions as the PKI not only imply problems based on constrained IoT devices. Further, they don't meet the security requirements in IoT authentication. "In IoT, things process and exchange data without human intervention. Therefore, because of this full autonomy, these entities need to recognize and authenticate each other as well as to ensure the integrity of their exchanged data. Otherwise, they will be the target of malicious users and malicious use. Due to the size and other features of IoT, it is almost impossible to create an efficient centralized authentication system." [22]

These are some weaknesses of centralized solutions as the PKI [1]:

- The single authorities centralizing identification act as possible central points of failure. If security breaches occur, for example due to wrongly implemented security profiles, then a single point of failure might compromise the security of its entire infrastructure.
- The PKI is the owner of the identities, leading to possibilities of abuse and failure. Identities might get shared among multiple users or identifiers of certain users can simply be removed as they are not owned by the users.
- By having centralized solutions, the availability of issued and revoked certificates in time is not guaranteed.
- Revocation of certificates implies additional security concerns. Having other services and lists running revoked certificates, these then add another point of failure by e.g. having a small delay on freshly revoked certificates.
- Centralized infrastructures as in the PKI can lead to scalability issues in systems that are used by thousands of nodes as it is possible in the IoT [22].

Decentralized Trust Model

In comparison to centralized solutions, a decentralized approach is coupling digital identities closer to the entities they are representing. Such a decentralized approach allows entities to validate and trust on each other's identities over the Internet. This is not only giving more control to the users on their digital identities, but it is also resulting in a complete recording of altered information concerning the identity. Additionally, it addresses security concerns of centralized solutions as the removal of single points of failure which may compromise the security of the entire system.
The decentralized approach introduced in this work is based on the web of trust that is removing the dependency on a single central authority because in the web of trust entities are trusting each other directly or through chains of trust.

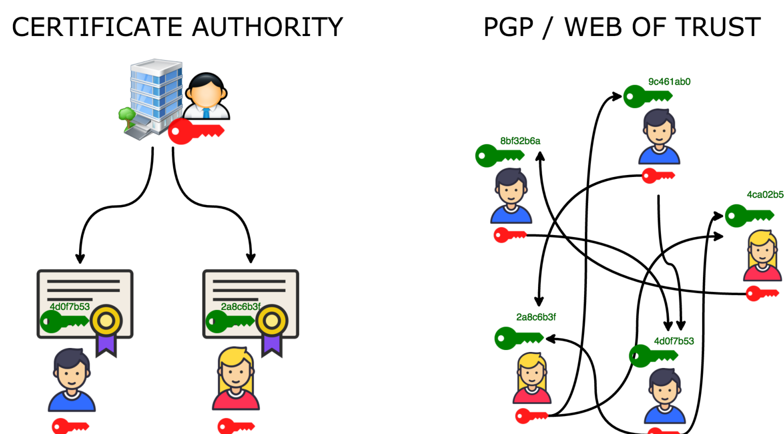


Fig. 2.11.: PKI compared to the Web of Trust, from [34]

This decentralized trust model is deployed using a smart contract running on an Ethereum blockchain decentralizing the required transactions. The deployment on a blockchain addresses further security requirements of IoT authentication presented in section 2.6.1..

Web of Trust

Trusting a public key in the web of trust means, that it was obtained directly from its owner (direct trust) or enough other users are trusting this key, in which one has trust. Pretty good Privacy (PGP) is used to present details of the Web of Trust. PGP was the first system to be introduced and used for the Web of Trust [15].

In PGP, users are signing public keys if they trust them. These signatures are called validity signatures. If no direct trust is given, then a PGP user only trusts in a public key if there are enough validity signatures from other participants, which are trusted by the user.

Fig. 2.12.: A Public Key Signing Meeting

Key Ring
Key rings are used to represent the trust in public keys between the PGP participants. Every PGP Participants is owning its own public key ring. This public key ring is not only containing their own public key, but also public keys of other PGP participants. Every entry on the key ring is containing the following fields:

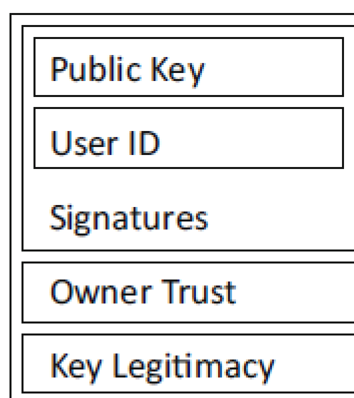


Fig. 2.13.: Single Key Ring Entry, from [15]

- A public key and its corresponding user ID that is identifying the owner of the public key.

- Validity Signatures of this public key together with the user IDs of the signers. The signature of the key ring owner has to be present in the signatures and further signatures may be added by other PGP participants.
  On key pair generation, PGP users create a self-signature being the signature of their own public key that needs to be present in every key ring containing the public key as entry. Before signing other's public keys, direct trust is expected to be pre-established. For example, key signing parties are organized where PGP users meet to establish direct trust by verifying each other's identity.
- Further, the key ring owner indicates with the owner trust its trust in the owner of the public key to trust other keys. This value is set by the key ring owner to each public key entry on its own key ring.
- The key legitimacy (key validity) indicates the trust of the key ring owner that the public key of the entry is belonging to the user of the User ID field.

2.6.1. Implications of a decentralized Trust Model

The introduction of a decentralized trust model running on a blockchain implicates several improvements compared to traditional centralized solutions. These improvements concern the security requirements and possible attacks on IoT authentication and communication.

- **Integrity**: Using blockchain technology is ensuring data integrity [14]. Further, the design of the smart contract allows to define the specific authorizations to modify its stored data.
- **Availability and Denial of Service**: The robustness and availability of the decentralized approach running on a blockchain is improved. Denial of service (DoS) attacks have less impact on the participants [13] and the transaction fees make it very costly to delay transactions to be included on the blockchain.
- **Scalability**: The usage of a blockchain relying on a peer-to-peer network provides high scalability [9].
- **Identification and Non Repudiation**: Transactions sent to the blockchain include the sender's signature identifying the sender and the sender cannot deny its transaction stored on the blockchain.
- **Sybil Attack and Identity Spoofing**: Sybil attacks can be avoided using a blockchain [30] and identities are bound to crypthographic keys. Therefore, an attacker has to possess the private key of the identity he wants to spoof.

The specific decentralized trust model used in this thesis is specified in section 3.4.6 and is designed to further address the security requirements and possible attacks in IoT authentication and communication.

2.7. Smart Contracts

With the recent attention and the therefore resulting fast development in blockchain technologies, programmed logic can be deployed decentralized using blockchain technologies. These new possibilities have emerged smart contracts which describe logical contracts that get executed on a blockchain. As the smart contracts are executing and decentral-

ized, they introduce new opportunities for transactions between untrusted parties. These properties remove the need of a third party that would establish trust between the participating parties.
As the logic that is defining the smart contract is decentralized, every participating party can examine and accept the conditions of smart contracts by themselves. Further, the immutability of the smart contracts logic introduces trust in its execution.

Three important characteristics are characterizing smart contracts [31].

Autonomy
After the deployment of the smart contract, its execution is not depending on its initializing user.

Self Sufficiency
Processing power or storage needed is marshaled by providing services not needing external intervention.

Decentralization
Being distributed and executed across the nodes in the network without any dependency on centralized servers.

Ethereum Virtual Machine (EVM)

The EVM is the virtual machine that is running the logic of the smart contract on an Ethereum blockchain and is executing stack-based bytecode. Therefore, the EVM introduces its own set of instructions, called "opcodes", that are defining the tasks that can be executed by the EVM. Furthermore, this set of instructions is Turing-complete.
The size of the Opcodes is defined as one byte. This allows up to 256 possible opcodes [37] that might be executed by the EVM. These opcodes further can be categorized:

Stack Manipulating
Opcodes that are manipulating the stack as POP or PUSH.

Arithmetic
Opcodes used for arithmetic operations such as ADD but also comparative opcodes such as GT (greater than) or logical opcodes such as AND.

Memory/Storage Manipulating
Operations used to manipulate either the memory or the storage. Memory manipulating opcodes would be MLOAD or MSTORE with the storage manipulating equivalents SLOAD and SSTORE.

Environmental
Environmental opcodes allow access on e.g. information about calls. These include the opcodes CALLER and CALLVALUE.

Program Counter / Halting
To support program counter operations, opcodes as JUMP or JUMPI are introduced.

Halting opcodes include the STOP and RETURN operations.

To efficiently store the needed opcodes, they are encoded and stored as bytecode having one byte that is allocated for every opcode. The bytecode then simply is split in its opcode containing bytes used for execution where every byte equals to two hexadecimal characters. Input data is extending the opcode byte by another byte if the input data is part of the opcode instruction.
Therefore, the two bytes 0x6001 would translate to the instruction PUSH1 (0x60) and the data being pushed (0x01).

Application Binary Interface

Deploying a smart contract to the blockchain requires a transaction to pass the smart contract's bytecode to the blockchain. The bytecode consists of two different parts.

- First, a part of the bytecode is used as a constructor in order to initialize the smart contract. The constructor bytecode is executed once and might be used e.g. to store variables on the smart contract's storage, which are needed by the runtime bytecode.
- The runtime bytecode is the second part defining the logic that runs on every transaction call of the smart contract. Additional information may be provided. Solidity appends the bytecode by bytes defining metadata that are not run as opcode by the EVM.

An Application Binary Interface (ABI) is used for the transactions concerning the smart contract. The ABI is describing the interface of the contract including its function names, as well as the input and output types.
To call a function on a contract, the function signature is derived by the four first bytes of a keccak256 hash of the function name including its inputs. If a function with the name "HelloWorld" would result in the signature hash "0x7fffb7bd", then a transaction would have to start with this hash being appended by additional data.
Additional data that is appending the signature hash of the transaction is getting processed as 32-byte words. Further, the first word following the signature hash is used to define the size of all following words as also needed for input data that is exceeding the word size of 32-bytes.

Gas

As the smart contracts are running on an Ethereum blockchain, every contract that is called will be executed by every Ethereum node in order to verify the results of the transaction. This would allow to slow down networks by introducing contract logic with computationally expensive operations. Therefore, a price to send transactions is defined, including calls of smart contract functions. Every opcode is assigned a fixed base cost. Further complicated opcodes are introducing an additional dynamic execution cost. Additionally, a base cost is added to every transaction. Values that are already stored on the smart contracts can be read without any costs.

2.7.1. Ethereum Gas Prices

The unit of Gas is connecting computational expenses to a resource consumption. This allows to distinctly define the computational costs of transactions without having a direct link to the current value of the cryptocurrency Ether itself that is used for the Ethereum blockchain.

The amount of Gas used for a transaction is not only including the computational expenses required for the computations done to process the transaction, but additionally a fee for the miner to include this transaction in the next mined block.

Gas Limit
Therefore, the Gas limit is introduced as the maximum amount of Gas that the sender of a transaction is willingly to spend to cover the computational cost of the transaction as well as the fee for the miner. Therefore, an increase of the Gas limit has an impact on the time required for a transaction to be included in new mined block. Transactions with higher fees as prioritized by the miners.

Gas Price
The sender of a transaction is not only including a Gas limit in its transaction. The sender is including as well a conversion that defines the currency rate between Gas and Ether. This conversion factor is defined as Gas price.

Ether
Ether is the cryptocurrency that is defining the monetary value of the Gas required to process transactions and to include them on the chain.
Ether is the resulting cryptocurrency by multiplying the used Gas with the Gas price that is included in the transaction.
The amount of Ethereum then can be exchanged to non-cryptocurrencies such as USD.

3 Implementation

3.1. Introduction **41**
3.2. Workflow **42**
3.3. Technologies **45**
3.3.1. Node.js 45
3.3.2. Smart Contract 47
3.4. Implementation **47**
3.4.1. Client 47
3.4.2. COSE Adapter 49
3.4.3. Authorization Server 50
3.4.4. Token Claim Key Translator 53
3.4.5. OSCORE Security Context Adapter 54
3.4.6. Smart Contract 55
3.4.7. Smart Contract API 58
3.4.8. Resource Server 59

3.1. Introduction

The practical part of this thesis is an implementation of an *Authorization Server* according to the ACE-OAuth framework. Instead of local authentication on the *Authorization Server* itself, a smart contract is introduced as a decentralized trust model implemented as a key ring. This approach allows decentralizing the authorization of token requests including a decentralized solution to allow *Resource Owners* to pre-establish authorization. Therefore, this implementation is a proof of concept for an ACE-OAuth implementation using a decentralized trust model.

Another goal of this thesis and its implementation is to provide a working base including the different specifications and technologies described in this thesis. Therefore, the implementation is not only a proof of concept but also an access point for working with the

related technologies to explore further scenarios more easily.

First, the workflow is described including the necessary steps for a *Client* to access a protected resource. Then, the considerations that were made for the implementation are presented. Finally, the implementation including the code that is of interest is described.

3.2. Workflow

The implemented ACE-OAuth workflow consists of several steps. A *Client* wants to access a protected resource. In the following example, the *Client* wants to access the resource stored on a temperature sensor. In order to access the protected resource, the *Client* needs to authorize itself to a *Resource Server* the resource is stored on.

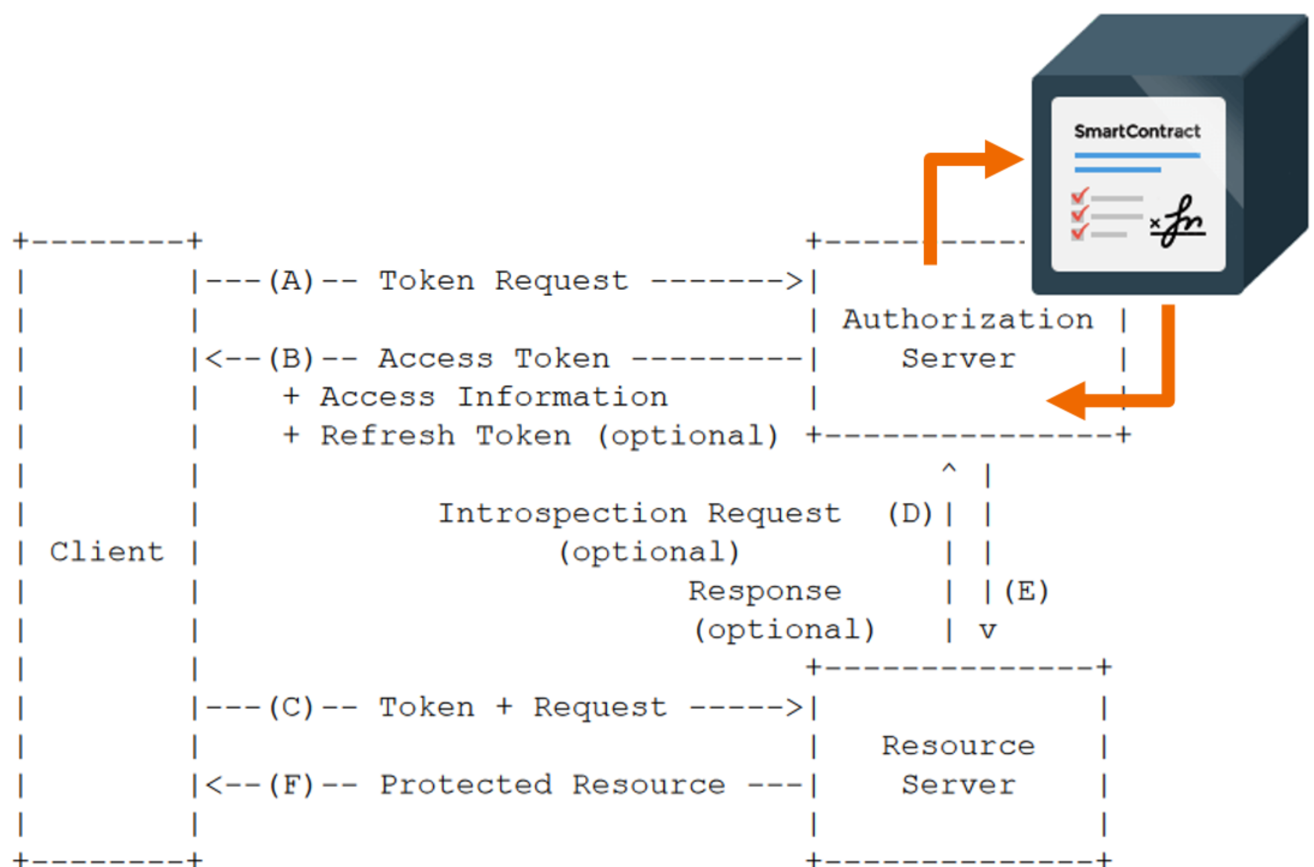


Fig. 3.1.: Workflow of the Implementation, adapted from [17]

Token Request

First, the *Client* needs to receive authorization to access the protected resource. The authorization is represented by a token which the *Client* is requesting from an *Authorization Server* sending a token request.
This token request to the *Authorization Server* consists of a CoAP message sent to the *Authorization Server*. This message includes a CBOR map with a set of claims defining

which resources the *Client* wants to access.
The token request is a CoAP POST message including a minimal set of claims needed. The claims are the audience, which is the identification of the *Resource Server*, as well as a public key of the *Client* that is needed for its identification. This public key is used to verify pre-established authorization by querying the decentralized trust model.
The *Client* is sending the claims as a COSE-Sign object including a signature of the claims. This is done in order to prove possess of the private key corresponding to the public key that is sent as a claim in the token request.

Pre-establishing Access

The *Client* needs the approval of the *Resource Owner* in order to access the owner's resource on a *Resource Server*. Therefore, the *Resource Owner* has to pre-establish authorization for the *Client* in order to have the *Client's* token request succeed.
A smart contract is used to enable the pre-establishing of access. In order to pre-establish authorization on the smart contract, a public key identifying a *Client* as well as an identifier of a *Resource Server* are stored on the smart contract. Further, a scope and an expiry time allow a more precise pre-provisioning of authorization.

Token Request Verification

The verification of the token request is done by the *Authorization Server*. Pre-established authorization doesn't have to be stored locally on the *Authorization Server* since the ACE-OAuth framework is extended by a smart contract which is responsible for providing a decentralized database of these pre-established accesses.
Therefore, the *Authorization Server* needs to know about the key ring addresses on the smart contract in order to query the pre-established authorizations.

The processing of the token request consists of three steps:

- First, the *Authorization Server* verifies that the public key included in the token request is matching the signature of the COSE-Sign object.
  Therefore, the *Authorization Server* verifies that the *Client* is in possession of the corresponding private key, if the verification of the signature is successful.
- After successful verification of the signature, the *Authorization Server* is querying the smart contract to verify if any access has been pre-established related to the requesting *Client*.
  Therefore, the public key and the audience which both are included in the token request are used as parameters.
- The smart contract returns an access object if authorization was pre-established on the key ring by a *Resource Owner*. The access object includes the scope and the expiry.

Receiving an access object from the smart contract denotes that the *Client* is authorized to access the requested resource.

Access Token Response

After successful verification, the *Authorization Server* is generating a pop access token to be sent back to the *Client*.
The pop token includes the public key of the *Client*. Therefore, the *Client* can later on send the token along another signature to allow a receiver of the token and the signature to proof the *Client's* possession of the key.
Further, identifiers for the *Resource Server* and for the *Authorization Server* are included in the token, as well as the scope and the expiry.
The pop access token is sent in a further CBOR map including the token itself but also the public key of the *Resource Server*. The public key of the *Resource Server* is required in order to allow the *Client* to generate a master secret needed for the derivation of an OSCORE security context. The master secret is generated by the Elliptic-Curve Diffie–Hellman protocol using its own private key and the public key of the *Resource Server*.

Token Upload

After the successful token response from the *Authorization Server*, the *Client* is in possession of the pop access token. Therefore, the *Client* is uploading the access token as a first step in order to access the protected resource. The token is sent in a COSE-Sign object in order to allow the *Resource Server* to proof the possession of the access token.
In a first step, the *Resource Server* proofs the possession of the token that was sent by the *Client*.
After successful verification, the *Resource Server* generates the same master secret as the *Client* did to derive a matching OSCORE security context. This is done analogously by using the ECDH protocol.

Resource Request

The *Client* and the *Resource Server* now can communicate over a secured channel as they have derived matching OSCORE security contexts. The resource request contains encrypted data in order to allow the *Resource Server* to verify, that their security contexts and the related keys are matching. This is done by creating and sending a COSE-Encrypt object.

Resource Response

The successful decryption of the received COSE-Encrypt content is the proof for the *Client's* authorization to access the requested resource. Therefore, the *Resource Server* responds to the resource request according to the authorization information received in the token.
The requested resource is sent protected as a COSE-Encrypt object using the derived keys from the security context.
The *Client* receives the requested resource as a CoAP response message with the COSE-Encrypt object that includes the encrypted resource data. Having the derived keys, the *Client* then can decrypt the COSE-Encrypt object and has successfully received the requested resource.

3.3. Technologies

In this section the technologies and third party libraries which were used for the implementation are presented.

3.3.1. Node.js

Node.js was used as the platform for implementing the roles of the ACE-OAuth framework required for the presented workflow, namely the *Authorization Server*, the *Client* and the *Resource Server*. Node.js allows fast prototyping of CoAP servers and is expandable by third party modules to meet additional requirements.

The following Node.js modules were used to meet the requirements for the implementation of the workflow.

coap
The coap module [42] is used to implement the CoAP servers representing the required roles of the ACE-OAuth framework. It is following the CoAP RFC specifications. The coap module further is using the coap-packet module in order to generate and parse CoAP packets.

```
1 const server = coap.createServer(coap_router)
2 server.listen(() => {})
```

List. 3.1: CoAP Server Instance

coap-router
The coap-router module [43] is used to define endpoints on the CoAP servers with the according handling of the incoming messages. The API is analog to known http routing modules as express-router.

```
1 coap_router.get("/Token", async (req, res) => {})
```

List. 3.2: Endpoint using coap-router

cbor
The cbor module [41] adds an API to encode and decode the data format. The module is following the CBOR RFC specification.

```
1 cbor.encode()
2 cbor.decode()
```

List. 3.3: CBOR Encoding and Decoding

cose-js
cose-js [44] implements the required COSE methods that are implemented following the COSE RFC specification. It is used to generate and verify COSE-Sign and COSE-Encrypt objects. It exposes an API to create and process these objects.

```
cose.sign.create(headers, payload, signer)
cose.sign.verify(cborSign, verifier)
cose.encrypt.create({p,u}, plaintext, recipientKey, options)
cose.encrypt.read(coseEncrypt, senderKey, options)
```

List. 3.4: COSE Sign and Encrypt

futoin-hkdf

The futoin-hkdf module [48] is used to derive the sender and receiver keys as well as the common IV during the establishing of the OSCORE security context. The implementation is following the RFC specification.

```
hkdf(masterSecret, L, info)
```

List. 3.5: HKDF for OSCORE

crypto

Crypto [45] is a Node.js built in module providing cryptographic functionality. The crypto module is used to generate elliptic curve keys. In addition, crypto is used for the Elliptic-Curve Diffie-Hellmann protocol in order to create a common master secret on the *Client* and the *Resource Server*.

```
ECDH_RS = crypto.createECDH('prime256v1');
ECDH_RS.generateKeys()
commonSecret = ECDH_RS.computeSecret(ECDH_Client.getPublicKey, null, 'hex');
```

List. 3.6: EC Key Generation and ECDH

Crypto is further used to create the signature and the keyhash which are later used as parameters calling the smart contract.

ec-pem

The key passed to the crypto sign function has to be in pem format. Therefore, the ec-pem module [46] is used to format the key.

```
var pemFormattedKey = ecPem(key, 'prime256v1');
```

List. 3.7: PEM Key Formatting

web3.js

The web3.js module [49] is required in order to enable communication between the node.js application and the smart contract deployed on an Ethereum blockchain.
Furthermore, the web3.js module is used in order to encode required parameters to the format accepted by the smart contract.

```
var encodedKey = dpki.web3.eth.abi.encodeParameters(
   ['uint256', 'uint256'], [publicKey1[0],publicKey1[1]])
```

List. 3.8: Smart Contract Parameter Encoding

ethereumjs-util
The ethereumjs-util module [47] is used similarly in order to match the same buffer formatting as it is used on the smart contract. This is done after the encoding using the web3.js module.

```
1 ethereumJSUtil.toBuffer(encodedKey)
```

List. 3.9: Matching Buffer Format

3.3.2. Smart Contract

Solidity

Solidity [50] is the high level language that is used in order to implement the smart contract. The code written in Solidity is compiled to bytecode that can be executed by the Ethereum Virtual Machine.

Truffle

Truffle [51] is the development environment used to test the smart contract that is written in Solidity. It is utilized for the compilation and deployment of the smart contract. Further, Truffle offers an interactive console for testing deployed smart contracts and was used for the first explorations of the contract's logic.

Ganache

Ganache [39] is part of the Truffle suite and allows running a local Ethereum blockchain by a single click. Therefore, Ganache is used to run the blockchain needed to deploy the smart contract. Additionally, the UI of Ganache provides information about the transactions made as well as about their Gas cost.

3.4. Implementation

In this section the implementation that is done in order to realize the workflow described is presented. Therefore, all of the previously mentioned technologies and modules were used. The implementation is accessible on GitHub [40].

3.4.1. Client

The *Client* is implemented as a mock object that is providing a simple interface to create the required CoAP requests and to process received CoAP responses.
Having the *Client* implemented as a mock object allows defining its behaviour and knowledge depending on the scenario.
Therefore, the constructor consists of the CoAP URIs of the *Authorization Server* as well as of the *Resource Server* that are used to build the requests. Furthermore, the constructor takes an EC private key to sign the resource request, as well as a CBOR encoded

claim map representing the requested resource in the token request.

```
class MockClient {
    constructor(uriAS, uriRS, privateKey, tokenRequestClaims) {
        this.uriAS = uriAS
        this.rsAdress = uriRS
        this.privateKey = privateKey
        this.tokenRequestClaims = tokenRequestClaims
}
```

List. 3.10: Client Constructor

In addition, the *Client* is offering functions to build CoAP requests. These CoAP requests then are used for testing the endpoints of the *Authorization Server* and the of *Resource Server*.

Token Request

The token request is targeting the "/token" endpoint of the *Authorization Server*. The request includes a CBOR encoded claim map defining the authorization requested by the *Client*.

```
GetTokenRequest() {
        const cborTokenReqClaims = cbor.encode(this.TokenRequestClaims)
        const coseSignTokenReq = await coseHelper.signES256(cborTokenReqClaims, this.
            privateKey)
        let coapTokenRequest = coap.request(this.uriAS + '/Token')
        coapTokenRequest.write(coseSignedTokenRequest)

        return coapTokenRequest
    }
```

List. 3.11: CoAP Token Request

Token Upload

The token upload is a CoAP message that includes the pop token, which the *Client* received from the *Authorization Server*. The payload is sent as COSE-Sign object in order to prove the possession of the private key related to the public key that is included in the token. The message then is sent to the "/authz-info" endpoint of the *Resource Server*.

```
GetAuthzUpload(accessToken) {
        const coseSign = await coseHelper.signES256(accessToken, this.privateKey)
        var authzInfoUpload = coap.request(this.uriRS + ':5000' + '/authz-info')
        authzInfoUpload.write(coseSign)

        return authzInfoUpload
    }
```

List. 3.12: Authz-info token upload

Resource Request

In order to have a successful request on the protected resource on the *Resource Server*, the *Client* first has to establish a security context according to OSCORE. Therefore, the *Client* is using the crypto module's Elliptic-Curve Diffie–Hellman function to generate a common master secret.

3.4.2. COSE Adapter

A COSE adapter is introduced to expose an interface that is wrapping the cose-js module. The COSE adapter is offering a function to create COSE Sign objects as well as a function to verify COSE Sign objects.

COSE-Sign Object Creation

The function to create the COSE-Sign object takes a payload and a private key as parameters. The private key is used as signer. Then, the function is building the header and signer objects according to the cose-js module requirements and calls the appropriate function of the cose-js module in order to generate the COSE-Sign object.
The function on the adapter is using the ES256 algorithm including the P-256 Elliptic Curve and the SHA-256 hash function in order to compute the signature.

```
1  signES256 = async (payload, privateKey ) => {
2      const headers = {
3          'p': { 'alg': 'ES256' },
4          'u': { 'kid': '11' }
5      }
6      const signer = {
7          'key': {
8              'd': Buffer.from(privateKey, 'hex')
9          }
10     }
11     return await cose.sign.create(headers, payload,signer)
12 }
```

List. 3.13: ES256 COSE Signing

COSE-Sign Object Verification

The function on the adapter used to verify the signature of a previously generated COSE-Sign object is taking a COSE-Sign object as well as the x and y coordinates of the public key as arguments. A verifier object including buffers of the passed x and y coordinates is built.
The verification function on the cose-js module then is called with the COSE-Sign object and with the built verifier object.

```
1  verifyES256 = async (coseSign, publicKeyX, publicKeyY) => {
2      const verifier = {
3          'key': {
```

```
4           'x': Buffer.from(publicKeyX, 'hex'),
5           'y': Buffer.from(publicKeyY, 'hex')
6       }
7   }
8   return await cose.sign.verify(Buffer.from(coseSign, 'hex'), verifier)
9 }
```

List. 3.14: ES256 COSE Sign Verification

3.4.3. Authorization Server

The main task of the *Authorization Server* is to process the token request of the *Client*. Therefore, the *Authorization Server* is performing three main steps:

1. Verification of the COSE-Sign object
2. Verification of the *Client's* Authorization
3. Generating and returning a pop access token

Token Endpoint

The "/token" endpoint on the *Authorization Server* is performing these three steps in order to provide the *Client* an access token if the verification was successful.

```
1  // Verification of the COSE Sign object
2  GET("/Token", (req, res) => {
3      const coseSignTokenReq = req[coseSignIndex]
4      const publicKeyX = decodedTokenRequest.req_cnf.COSE_Key.x
5      const publicKeyY = decodedTokenRequest.req_cnf.COSE_Key.y
6      verifyCoseSign(
7          coseSignTokenReq,
8          publicKeyX,
9          publicKeyY)
10
11 // Verification of the Client's Authorization
12     .then((tokenReqClaims) => {
13         const aud = tokenReqClaims.aud
14         return verifyAccess(
15                     clientPubX,
16                     clientPubY,
17                     aud)
18     })
19
20 // Generating and returning a pop access token
21     .then((access) => {
22         return createCWT(
23             tokenReqClaims,
24             access)
25     })
26 })
```

List. 3.15: Token Request Endpoint

COSE-Sign Verification

The incoming payload of the token request is a CBOR encoded COSE-Sign object.
First, the COSE-Sign object is accessed from the request.
Afterwards, it is decoded from the CBOR format. The decoded data contains the claims that define the access requested in this token request.
The claims include the public key of the *Client* related to the *Client's* private key that was used to create the COSE-Sign object. This is done to provide a pop token.

```
1 const coseSignTokenReq = req.payload
2 const decodedTokenReq = await cbor.decode(coseSignTokenReq)
3 const tokenReqClaims = await cbor.decode(decodedTokenRequest.value[indexClaims])
4 const clientPubX = tokenReqClaims.req_cnf.COSE_Key.x
5 const clientPubY = tokenReqClaims.req_cnf.COSE_Key.y
6 coseHelper.verifyES256(
7               coseSignTokenReq,
8               publicKeyX,
9               publicKeyY)
```

List. 3.16: Signature Verification

Verification of the Authorization

The *Authorization Server* verifies if the *Client* received authorization from a *Resource Owner* in order to access a protected resource. This step is performed after successful verification of the COSE-Sign object.
The audience included in the claims as well as a sha256 hash of the provided public key points are passed as parameters to the smart contract in order to query pre-established authorization bound to the *Client's* public key. Therefore, the public key of the *Client* first is encoded and hashed using modules to receive a key hash that is formatted equivalent to the key hashes that are computed on the smart contract using the available sha256 function.
If a *Resource Owner* pre-established authorization on the requested resource for the *Client*, then the smart contract is returning an access object that includes the expiry and scope for the authorization.
The call of the function on the smart contract is additionally taking the pre-established address of the key ring as well as the address of the caller which is the *Authorization Server*.

```
1  verifyAccess(x, y, aud) {
2     var publicKey = [
3         '0x' + x,
4         '0x' + y
5     ];
6     var encodedKey = dpki.encodeParameters(
7                               ['uint256', 'uint256'],
8                               [publicKey[0], publicKey1[1]])
9     var keyHash = '0x' + crypto.createHash('sha256')
10                                .update(ethereumJSUtil.toBuffer(encodedKey))
11                                .digest('hex')
12
```

```
13    var access = await dpki.verifyAccess(
14                                dpki.accounts[indexKeyRing],
15                                keyHash,
16                                aud,
17                                dpki.accounts[indexAuthorizationServer])
18
19    return access
20 }
```

List. 3.17: Access Verification

Pop Token Response

The last step required to process the token request includes the creation of the pop access token. This token then is returned to the *Client* after successful verification of its token request.
The claim map received in the token request as well as the access object received from the smart contract are required for generating the pop access token for the *Client*.

First, the COSE-Key map of the token request is accessed which is including the public key of the *Client* that is related to the requested access.
Afterwards, the claim map for the access token is created. The audience which is identifying the *Resource Server* was received within the token request. The expiry and the scope are contained in the access object received from the smart contract. The iss is the identifier of the *Authorization Server* that is issuing the token. Finally, the COSE-Key claim map accessed before is assigned to the cnf claim of the token. This cnf claim is later required in order to allow proof of possession of the token as it contains the public key of the *Client* that was received within the token request.

The verbose claim keys then are translated to corresponding integers in order to reduce the size of the claim map. This claim map with the translated keys is CBOR encoded.
The created pop access token then is included in a further CBOR map together with the public key of the *Resource Server*. The *Authorization Server* includes the public key of the *Resource Server* so that the *Client* can perform the Elliptic-Curve Diffie-Hellman protocol to generate the master secret that is required to establish a secured channel to the *Resource Server*.

Finally, the payload of the token response consists of a COSE Sign object generated by the *Authorization Server*.

```
1  createCWT(tokenReqClaims, access) {
2
3      const coseKey = tokenReqClaims.req_cnf.COSE_Key
4      const tokenClaims = {
5          aud: tokenReqClaims.aud,
6          iss: 'exampleAS',
7          exp: access.expiry,
8          scope: access.scope,
9          cnf: {
10             COSE_Key: coseKey
```

```
        }
    }

    const translatedTokenClaims = await cwtHelper.translateClaims(tokenClaims)
    const cborTokenClaims = await cbor.encode(claimMap)

    const resPayload = {
        access_token: cborTokenClaims,
        rs_cnf: {
            COSE_Key: {
                kty: 'EC',
                crv: 'P-256',
                x: preestablishedKeys[indexResourceServer].x,
                y: preestablishedKeys[indexResourceServer].y
            }
        }
    }
    const cborResPayload = await cbor.encode(resPayload)
    let coseSignResponse = await coseHelper.signES256(cborResPayload, privateKey)

    return coseSignResponse
}
```

List. 3.18: Access Token Response

3.4.4. Token Claim Key Translator

In order to reduce the size of the access token generated by the *Authorization Server*, its claim keys are translated into specified integers. Therefore, a translator object is implemented offering a function to translate claim keys of claim maps.

```
translateClaims(message) => {
    var claimMap = new Map()
    await translateKeys(message, claimMap)

    return claimMap
}
```

List. 3.19: Claim Key Translation

A registry is needed to define the conversion from the verbose claim keys to a related unique integer. The registry is defined on different levels of nestings on the claim map. This allows to use the same integer for different verbose claim keys.

```
let claims = new Array()
claims['root'] = {
    iss: 1,
    sub: 2,
    aud: 3,
    exp: 4,
    nbf: 5,
    iat: 6,
    cti: 7,
```

```
10     cnf: 8,
11     scope: 9
12 }
13 claims['cnf'] = {
14     COSE_Key: 1
15 }
16 claims['COSE_Key'] = {
17     kty: 1,
18     crv: -1,
19     x: -2,
20     y: -3
21 }
```

List. 3.20: Claim Key Registry

The translation algorithm is iterating through all keys that are included in the claim map. If the current key value is an object, then a new map is instantiated and the algorithm is doing a recursive calls having the next level of nesting. If a claim key is not yet defined in the registry, then it is not getting translated and keeps its initial value.

```
1      var currMap = new Map()
2      let claimObject = claimDict in claims ? claims[claimDict] : claims['root']
3      for (var key in obj) {
4          // Test if the current key is defined in the actual dictionary
5          if((Object.keys(claimObject).toString()).includes(key.toString())){
6              if(isObject(obj[key])){
7                  currMap = new Map()
8                  map.set(claimObject[key], currMap)
9                  translateKeys(obj[key], currMap, key.toString())
10             } else {
11                 map.set(claimObject[key], obj[key])
12             }
13         }
14         else{
15             if(isObject(obj[key])){
16                 currMap = new Map()
17                 map.set(key, currMap)
18                 translateKeys(obj[key], currMap, key.toString())
19             } else {
20                 map.set(key, obj[key])
21             }
22     }
```

List. 3.21: Key Translation

3.4.5. OSCORE Security Context Adapter

An OSCORE security context adapter is introduced to simplify the derivation of a common security context. The constructor of the adapter is accepting all parameters that might be used to derive the full security context. Additionally, the constructor defines default parameters if the OSCORE specification has defined a default value for the specific parameter. The mandatory parameters are the IDs of the sender and receiver as well as the master secret.

```
class OscoreSecurityContext {
    constructor(
            senderID,
            receiverID,
            masterSecret,
            masterSalt = '',
            idContext = null,
            aeadAlg = 10,
            hkdfAlg = 'SHA-256')
        {
        this.masterSecret = Buffer.from(masterSecret, 'hex')
        this.senderID = senderID,
        this.receiverID = receiverID
        this.masterSalt = Buffer.from(masterSalt)
        this.idContext = idContext
        this.aeadAlg = aeadAlg,
        this.hkdfAlg = hkdfAlg

        this.deriveFullContext()
        }
```

List. 3.22: Security Context Constructor

The security context is derived directly at the end of the constructor. The deriveFull-Context function is deriving the keys for the sender and receiver as well as the common IV. At least the IDs and the master secret have to be passed to the constructor in order to have enough parameters to derive the OSCORE security context.

```
deriveFullContext() {
        this.senderKey = deriveSenderKey()
        this.receiverKey = deriveReceiverKey()
        this.commonIV = deriveCommonIV()
    }
```

List. 3.23: Deriving the Security Context

The three derivation functions are analog. First, the necessary info object is generated. Then, this info object is passed to the hkdf function from the futoin-hkdf module along with the master secret and the length required for the specific parameter that is computed.

```
deriveSenderKey() {
    let cborInfo = this.buildSerializedInfo('Key', this.senderID)

    return this.runHKDF(cborInfo, 16)
}
```

List. 3.24: Derivation of the Sender Key

3.4.6. Smart Contract

Solidity Implementation

The smart contract is based on the key ring model. First, data structures are introduced which are shared among the smart contract in order to model a key ring.

First, two mappings that store all public keys as well as their revocations are defined on the smart contract. Defining these mappings at that level allows storing this information at a single point. Therefore, some concerns of Public Key Infrastructures are resolved as the delay of learning about freshly revoked keys.

```
mapping (bytes32 => address) public publicKeys;
mapping (bytes32 => bool) public revokedKeys;
```

List. 3.25: Public Key and Revocation registry

The "publicKeys" mapping is used in order to register public keys on the smart contract that further can be used by *Resource Owners* in order to pre-establish authorization. It maps a keyhash of a public key to the address of participant that knows about the corresponding private key. The "revokedKeys" mapping maps the same keyhash to a Boolean that is defining if a certain key is revoked.

Adding a Public Key

Adding new keys introduces concerns. For example, public keys can be locked if there is no protection on the "publicKeys" mapping. These concerns are resolved by including a third party implementation [38] of an ECDSA signature verification algorithm that is required to success, in order to store a new public key on the key ring. The function that is verifying the signature takes a hash of the public key, the signature, as well as the public key as input.

```
function validateSignature(
                    bytes32 keyHash,
                    uint[2] memory rs,
                    uint[2] memory Q) public pure
    returns (bool)
```

List. 3.26: Signature Validation

First, the public key consisting of the x and y coordinates is encoded to a byte array that is sha256 hashed. The *Authorization Server* is including the hash of the public key in order to ensure that it is using the same encoding and hashing as the smart contract. If both hashes are identical it is further verified that the key is not yet stored on the contract. If the key is not yet stored its signature is verified. After successful verification the key is added to the "publicKeys" mapping.

```
addKey(bytes32 keyHash, uint[2] memory rs, uint[2] memory Q)
    public returns (bytes memory, address,uint[2] memory, uint[2] memory){
        bytes memory b = abi.encodePacked(Q[0],Q[1]);
        bytes32 hashQ = sha256(b);
        require(hashQ == keyHash, 'Hash missmatch with provided key Q');
        require(publicKeys[hashQ] == address(0), 'key existing');
        require(validateSignature(hashQ, rs, Q) == true, 'signature missmatch');
        publicKeys[keyHash] = msg.sender;

        return (b, publicKeys[keyHash], rs, Q);
```

List. 3.27: Adding a Public Key to a Key Ring

Public Key Revocation

The revocation of keys makes use of the public key storage as they are connected to the address of the transaction sender. Therefore, this address can be used in order to allow a participant to revoke stored public keys. The revocation of a key is requiring that the key is stored in the "publicKeys" mapping and that the sender of the revocation transaction is the same as for the storing of the key.

```
revokeKey(bytes32 keyHash) public{
    require(publicKeys[keyHash] != address(0), 'key not registered');
    require(publicKeys[keyHash] == msg.sender, 'no permit to revoke');
    revokedKeys[keyHash] = true;
}
```

List. 3.28: Key Revocation

Keyring Structure

The keyring itself is implemented as a struct. The keyring struct includes the address of the key ring owner. Further, it contains addresses of keyrings of other entities in which the key ring owner trusts to sign other keys. Finally, the struct contains a mapping to store pre-established access. "KeyAccess" is mapping a keyhash to an audience identifier which then maps to an access object.

```
struct KeyRing{
    address ringOwner;
    address[] trustedRings;
    mapping (bytes32 => mapping(string => Access)) keyAccess;
    }
```

List. 3.29: Key Ring

Keyring Creation

A transaction is needed in order to create a new keyring. Each key ring is stored in a further mapping from participant addresses to keyrings. The function that is creating new keyrings adds the address of the caller to the "keyRings" mapping and verifies that the mapping of this address did not exist before.

```
mapping (address => KeyRing) public keyRings;

createKeyRing() public{
        KeyRing storage keyRing = keyRings[msg.sender];
        require(keyRing.ringOwner == address(0), 'keyRing existing');
        keyRing.ringOwner = msg.sender;
}
```

List. 3.30: Key Ring Creation

Access Object

The Access object is a simple struct that is stored on the "keyAccess" mapping on the keyrings. It consists of a scope and an expiry field. The scope defines the *Client's* authorizations on the resources stored on the specific *Resource Server*. Then, the expiry field is defining the expiration date of this pre-established access.

```
struct Access{
      string scope;
      uint expiry;
   }
```

List. 3.31: Access

Pre-establishing Access
Only the owner can pre-establish access objects on its own keyring.
The function that is adding access objects is therefore requiring that the sender is the owner of the key ring and that the according key is not revoked yet.

```
giveAccess(bytes32 keyHash, string memory aud, string memory scope, uint expiry)
   public{
       KeyRing storage keyRing = keyRings[msg.sender];
       require(keyRing.ringOwner == msg.sender, 'not keyRing Owner');
       require(revokedKeys[keyHash] == false, 'key is revoked');
       keyRing.keyAccess[keyHash][aud] = Access(scope, expiry);
   }
```

List. 3.32: Provisioning Access

In addition, the contract is offering functions to alter the values of the expiry as well as the scope on previously stored access object.

Chain of Trust
In order to enable a chain of trust the "trustedRings" mapping is defined on the keyring struct. If a keyring owner added an address of another keyring to the "trustedRings" mapping it means, that the keyring owner trusts the related keyring owner. Three functions are provided to enable querying chains of trust:

- A function allowing the owner of a keyring to add further keyring addresses to the "trustedRings" mapping.
- In addition, another function allows deleting keyring addresses that were previously stored on the "trustedRings" mapping.
- The third function is returning the number of addresses that is stored in the mapping. This value can be used to loop over the existing addresses as they have to be accessed with indices.

3.4.7. Smart Contract API

The web3 node.js module is first used to deploy the contract. After the successful deployment of the smart contract its functions can be called using the same web3 module.

First, the web3 module is instantiated with the address of the blockchain, where the contract will be deployed to. The address is accessible on the Ganache UI.
The web3 module offers a function to deploy smart contracts which requires their Application Binary Interfaces (ABI). These Interfaces are created during the compilation of the Solidity implementation. The web3 module requires the address of the blockchain as

well as the ABI to create a Contract object. This generated Contract object allows the *Authorization Server* to communicate with the smart contract.

```
deployContract() {
    const contract = await new this.web3.eth.Contract(
        applicationBinaryInterface, { data: abi, from: accounts[0], gas: amountOfGas})

    return new Promise((resolve) => {
      contract.deploy().send()
        .on('receipt', (receipt) => {
          var contractAddress = receipt.contractAddress
          const DPKI = new this.web3.eth.Contract(this.abi, contractAddress)
          resolve(DPKI)
        })
    })
}
```

List. 3.33: Contract deployment

Further, an adapter is introduced in order to facilitate the communication between the *Authorization Server* and the deployed smart contract. The adapter is wrapping the web3 functions and reduces the amount of required parameters. These wrapped web3 functions make calls on the instantiated Contract object. Here only the wrapper for the addKey function is shown as the further wrappers are implemented analogously.

```
addKey(keyHash, signature, publicKey, sender) {
    return new Promise((resolve) => {
      this.contract.methods.addKey(keyHash, signature, publicKey)
        .send({ from: sender, gas: amountOfGas}) })
        .then((hash) => {
          resolve()
        })
    })
}
```

List. 3.34: Smart Contract function wrapper

3.4.8. Resource Server

The *Resource Server* is implemented as an independent CoAP server. Two endpoints are defined on the *Resource Server* in order to test the implementation.

authz-info

The authz-info endpoint is used by *Clients* in order to upload access tokens that are related to resources on the *Resource Server*. The *Resource Server* is first performing a proof of possession for the received pop token. Therefore, the *Resource Server* is verifying the COSE Sign object it received from the *Client* that is containing the access token.
If the verification of the COSE Sign object was successful, then the pop token is getting stored in a local registry for *Client*-tokens.

```
1  GET("/authz-info", async (req, res) => {
2      var signedTokenPayload = req.payload
3      var decodedTokenPayload = await cbor.decode(signedTokenPayload)
4      var decodedToken = await cbor.decode(decodedTokenPayload.value[tokenIndex])
5      var clientPubX = decodedToken.get(pubX)
6      var clientPubY = decodedToken.get(pubY)
7
8      coseHelper.verifyES256(signedTokenPayload, clientPubX, clientPubY)
9      .then((buf) => {
10         res.end()
11     })
12 })
```

List. 3.35: authz-info Endpoint

The *Resource Server* is in possession of all parameters that are required to derive an OSCORE security context. Therefore, the master secret is computed first by using the crpyto module and its Elliptic-Curve Diffie-Hellman function. Then, the security context is derived by using the OSCORE adapter.

Resource Endpoint

The Resource Requests of the *Client* is a COSE Encrypt object. After the derivation of the security context the *Resource Server* is able to decrypt the COSE-Encrypt object and therefore to validate the resource request from the *Client*. The *Resource Server* then returns the protected resource if the authorization of the *Client* was verified successfully.

4 Results

In this section all CoAP messages and blockchain transactions are presented that were required to have a *Client* accessing a protected resource as defined in the workflow according to the ACE-OAuth framework.
Then the Gas prices that are required for the transactions defined in the workflow are assessed. Further, the Gas prices are converted to USD by using actual exchange rates of the Ethereum cryptocurrency Ether in order to examine, if the resulting prices allow a realistic usage of the decentralized trust model.

4.1. Workflow Results

Testing the Implementation

An integration test is used to validate the exchanged messages and therefore, that the implementation of the specified scenario is working as intended.
First, the integration test prepares the required components in order to start sending and receiving CoAP messages.

1. The smart contract is deployed by using the web3 module.
2. The cryptographic keys are generated for the *Client*, for the *Authorization Server* and for the *Resource Server*. Then, the generated keys are stored in a global variable to imitate pre-provisioning.
3. The authorization of the *Client* is pre-established on the smart contract. Therefore, the public key that was generated for the *Client* is added to the smart contract. Then, a keyring is created for the *Resource Owner* in order to store access for the *Client* on the keyring.
4. The CoAP servers (the *Authorization Server* and the *Resource Server*) are started then.
5. Finally, the mock *Client* that is used by the integration test in order to exchange messages with the started CoAP servers is instantiated.

Token Request

The first step for the *Client* is to request an access token from the Authorization in order to access a protected resource. Therefore, the *Client* is sending a CoAP message to the

"/token" endpoint of the *Authorization Server* that is including a COSE-Sign object containing a set of claims representing the desired Authorization. The set of claims includes the generated public key of the *Client* and is CBOR formatted.

```
1 0xA3636175647674656D7053656E736f72496E4C6976696E67526F6F6D69636C69656E745F69646963C...
```

List. 4.1: CBOR Formatted Token Request Claims

This CBOR map in hexadecimal notation can be translated to the CBOR diagnostic notation in order to assess the claims sent by the *Client*.

```
1  "aud": "tempSensorInLivingRoom",
2  "client_id": "client123",
3  "req_cnf":
4  {
5      "COSE_Key":
6      {
7          "kty": "EC",
8          "crv": "P-256",
9          "x": "498168833a74d0da9273261d05738df5a2dbc6e354d3dffc0d8e1c8e5046ac04",
10         "y": "0c3ca7d6cd60ff18fa1ae727d73f3121e94a7e5e8a5b9b48517c40f19226c878"
11     }
12 }
```

List. 4.2: Token Request in Diagnostic Notation

In this scenario the scope of the requested access is not included in the token request sent by the *Client*, as the scope and the expiry are pre-established by the *Resource Owner*. This is done during the preparation of the integration test.

Access Verification

The first action of the *Authorization Server* is to verify the COSE Sign object that was sent as the payload in the token request. Afterwards, the *Authorization Server* is computing a hash of the *Client's* public key, if the verification of the COSE-Sign object was successful.
The computed hash of the public key then is encoded to the format that is used by the smart contract. For this encoding, the *Authorization Server* uses the web3 module.
The formatted keyhash is then sent to the verifyAccess function on the smart contract in order to verify if a *Resource Owner* pre-established access for the *Client*. The additional data appended to the function call are the audience that was included in the token request claim map, as well as the computed keyhash in the according format. Both arguments are required for the access object mapping on the keyring.

As a relating access object was stored on the keyring during the preparation of the test, it is now returned from the smart contract to the *Authorization Server*. The access object includes the scope that defines which specific resource the *Client* may access, as well as the expiry of this access.

```
Access
{
    scope: 'temp_g',
    expiry: '1000'
}
```

List. 4.3: Pre-established Access returned from the Smart Contract

Token Response

The *Authorization Server* is generating a pop access token after receiving a related access object from the smart contract. At this point the access token is representing the authorization that the *Client* received from the *Resource Owner*.
The public key of the *Client* that was included in the token request is now stored in the pop token in order to enable proof of possession of the token. Further, the information of the access object and identifiers for the *Authorization Server* and for the *Resource Server* are included in the token.
A further CBOR map is created to store the generated pop access token, as well as the public key of the *Resource Server*. This CBOR map is sent as a COSE-Sign object. This allows the *Client* to verify that the token was received by the *Authorization Server*.

```
"access_token":"pQN2dGVtcFNlbnNvckluTGl2aW5nUm9vbQFpZXhhbXBsZUFTBGQxMDAwCWZ0ZW1wX...",
"rs_cnf":
{
    "COSE_Key":
    {
        "kty": "EC",
        "crv": "P-256",
        "x": "989d96f139fb8552e764bf169e07d0b0946e83c38654e9dac3a53697037759af",
        "y": "b75ad56814c557ccbf6fb5e63af569ea6ec4e00fa47c56ed94a74aba3ad64780"
    }
}
```

List. 4.4: Payload of the COSE Sign Token Response

The "access token" claim is containing the pop access token. This access token is a CBOR encoded map. In order to reduce the size, the access token, the claim keys are translated to their corresponding numeric identifiers. The Claim Key Translator object is used for the translation of the claim keys.

```
3: "tempSensorInLivingRoom",
1: "exampleAS",
4: "1000",
9: "temp_g",
8:
{
    1:
    {
        1: "EC",
        -1: "P-256",
        -2: "498168833a74d0da9273261d05738df5a2dbc6e354d3dffc0d8e1c8e5046ac04",
        -3: "0c3ca7d6cd60ff18fa1ae727d73f3121e94a7e5e8a5b9b48517c40f19226c878"
```

```
13     }
14 }
```

List. 4.5: POP Access Token

Token Upload

The *Client* then creates another COSE-Sign object that is including the access token received from the *Authorization Server*. This COSE-Sign object is created so that the *Resource Server* can proof the *Client's* possession of the token.
This COSE-Sign object is then sent by the *Client* to the "/authz-info" endpoint of the *Resource Server*.

Establishing a Secured Channel

After the successful upload of the access token, the *Client* and the *Resource Server* are in possession of the required parameters to derive matching cryptographic keys.

The master secret is derived by using the Elliptic-Curve Diffie-Hellman protocol that is implemented in the crypto module. Both security contexts are then derived using the OSCORE Security Context. The *Resource Server* is connecting the security context to the access token that is including the public key of the *Client*. This is relating the Sender and Recipient key of the security context to the authorization of the *Client*.

Resource Request

The last step of the integration test is the resource request of the *Client* resulting in a resource response from the *Resource Server*. The resource request is sent as a COSE Encrypt object that is encrypted using the Sender Key of the derived security context.

The *Resource Server* knows that the *Client* is authorized to access the protected resource after the successful decryption of the COSE-Encrypt object. Then, the recipient key of the derived security context is used for the decryption of the message that was encrypted using the sender key.
After successful verification of the *Client's* authorization, the *Resource Server* sends the requested resource to the *Client*. The *Client* receives a CoAP message containing the requested temperature that is stored on the *Resource Server*.

```
1 Resource
2 {
3     temperature: 30
4 }
```

List. 4.6: Resource Response

4.2. Gas Prices

In this section an assessment of the blockchain transaction prices is presented. These prices are important to assess possible scenarios. The amount of Gas spent is therefore converted to its current value in USD.
First, the required Gas as well as the defined price for the Gas have to be defined to calculate the monetary effort in order to alter the state of the smart contract running on the blockchain.
The amount of Ether spent on the transactions is then calculated by multiplying the amount of Gas spent with the Gas price. This calculation is done for the transaction that deployed the smart contract as well as for the transactions used in the integration test.

The open source web tool ETH Gas Station [35] is used to calculate the required amount of USD that would be necessary to deploy and use the smart contract as in the implementation. This web tool is offering several statistics about the Ethereum blockchain and current transactions.

Transaction Processing Time

Additionally, it is important to consider the required amount of time that is needed in order to have a transaction included on a newly mined block. The amount of required time is related to the fee included for the miner. Therefore, the amount of USD spent for a transaction is higher if the transaction should be included on the blockchain faster. The web tool recommends Gas price depending on the desired time required to be included on the blockchain.

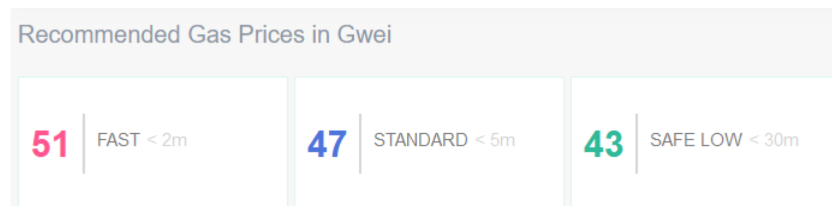


Fig. 4.1.: Recommended Gas Prices for Processing Times

These recommendations are based on statistics that are as well accessible on the web tool. One of these statistics is relating the Gas price of a transaction with the chance that it gets included in a future block. The Gas price is listed in the first column using the unit Gwei that is a billionth of an Ether and the second column lists the percentage of the previous 200 blocks that accepted transactions related to the Gas price.

Txpool Data At Block 10588726

Gas Price (Gwei)	% of Last 200 Blocks Accepting	% of total transactions mined in last 200 blocks	#Tx at/above in txpool	Median Age (blocks)	Total Seen 5m	Pct Mined 5m	% of Tx Unmined > 5min	Total Seen 30m	Pct Mined 30m	% of Tx Unmined > 20min
32	8.2	0.9	241	44.5	12	0	83	2	0	100
38	14.3	4.6	207		---	---	---	1	0	0
40	21.4	11.4	199		---	---	---	1	100	0
42	35.7	12.6	193		1	0	0	---	---	---
43	45.9	13.7	186		---	---	---	2	50	0
45	52	15.4	185		---	---	---	1	0	0
46	57.1	16.9	185		1	0	0	---	---	---
47	66.3	19.4	183		1	0	0	2	100	0
48	76.5	20.3	177		1	0	0	1	0	0
49	76.5	21.4	171		---	---	---	1	100	0
50	82.7	39.1	171	0	9	22	0	1	0	0
52	88.8	40.9	129		2	50	0	1	100	0
53	91.8	43.7	129	0	---	---	---	1	100	0

Fig. 4.2.: Gas Prices related to the Acceptance of Transactions

Not in every scenario it is important to have the access pre-established as quick as possible. Therefore, a lower Gas price can be chosen for the calculations of the final costs.

Calculated Prices

The calculations for the prices are made on the 19th of July 2020 using the following values:

- 1 Ethereum = 233.97 USD = 219.61 CHF
- Gas Price: 43 Gwei

These conversion rates are used in the calculator [36] of the ETH Gas Station web tool in order to calculate the prices of the transactions. The result then can be converted to USD.

The prices in USD are calculated for every transaction that was required for the implementation of the workflow. The amount of Gas spent is accessible in the Ganache UI. This amount of Gas spent then is multiplied with the parameters defined above. The following list presents the amount of Gas and the calculated prices in USD for the transactions.

- Contract Deployment: Gas = 3'504'048, USD = 36.073
- Function "addKey": Gas = 1'129'781, USD = 11.631
- Function "createKeyRing": Gas = 42'160, USD = 0.434
- Function "trustRing": Gas = 63'979, USD = 0.659

- Function "giveAccess": Gas = 69'292, USD = 0.713
- Function "verifyAccess": Gas = 0, USD = 0

The deployment of the smart contract is the most expensive transaction. The amount of Gas was used for the deployment without any change of state. Therefore, no public keys and no key rings are initially deployed. The price of nearly 40 USD for the deployment of an infrastructure that is enabling a decentralized trust model is very low compared to the price for centralized servers including the maintenance of the hardware and the software, as well as the costs to run them.

The "addKey" function, which is storing a key on the smart contract, is verifying the signature on the smart contract itself and is therefore more expensive than the other transactions. Storing a public key on the smart contract enables the participation in the decentralized trust model. Therefore, 12 USD to add a key are the initial costs to participate. These initial costs can be reduced by a significant amount by simplifying the smart contract.

All further transactions that are required in order store and access authorization cost less than 1 USD with a Gas price that requires about 30 minutes to be included on the blockchain.
Higher Gas prices may be paid to reduce the required amount of time below 5 minutes increasing the costs by less than 10%. Therefore, scenarios requiring transactions that are included quickly can be realized without having inaccessible prices.

5 Outlook

The implementation made in this thesis is providing an extensible structure to implement different scenarios and adapt the existing logic. As one of the main goals is the provisioning of such a structure for the ACE-OAuth framework, several outlooks are of interest. This includes an access point for testing different implementations in order to explore new scenarios for IoT authentication.
In this chapter several ideas for future work are discussed. These ideas build on top of the implementation of this thesis.

5.1. ACE-OAuth Module

With the currently available Node.js modules, an implementation of an ACE-OAuth framework protocol is feasible as it was made for the implementation in this thesis. But there are still several smaller components which have to be connected in order to realize a scenario that that is using the ACE-OAuth framework. A module integrating all distinct modules to provide a simple interface would be desirable. This would allow faster prototyping of scenarios that build on the ACE-OAuth framework in order to explore further possibilities. A single module could also collect contributions in a single point.

Two approaches could be realized concerning modules in order to increase their accessibility:

1. Several adapters have been implemented for the implementation of this thesis in order to simplify the usage of specific protocols. In one approach these adapters could be extended and further adapters could be added. These adapters then could be collected and combined in a single module offering all the functionality required for certain scenarios having the module as single entry point to communicate with the specific modules. This includes for example the derivation of an OSCORE security context or the generation of CBOR web tokens as well as the translation of the claim keys.
2. In another approach, the existing modules could be extended in order to offer the required functionality for implementing scenarios that build on the ACE-OAuth framework.

5.2. Further Scenarios

Further scenarios can be implemented using the implementation of this thesis as structure. The introducing of a decentralized trust module is enabling numerous possible scenarios. It is of interest to have distinct scenarios explored in order to verify their practicability and to discover further scenarios.

Additionally, a scenario could be deployed in a real world example containing constrained nodes to build an ecosystem using decentralized trust models. It would be of interest to explore new and different requirements related to the constrained nodes.

5.3. Smart Contract

When exploring smart contracts, it is a big challenge to compile and deploy them. A further issue is to have a certain runtime being able to communicate with the deployed smart contract.
Therefore, the approach for the implementation of this thesis was to simplify the compilation, the deployment and the communication as much as possible. This approach could be improved and the different solutions could be combined in a single node.js module in order to make smart contracts more accessible.

As a future work, more trust models could be modeled using smart contracts to further explore the strengths and weaknesses of different decentralized implementations compared to services offered by third parties.

An increase of the accessibility of the ACE-OAuth framework as well as of smart contracts could result in more interest in these technologies resulting in a higher participation.

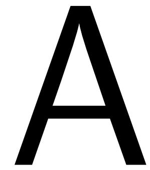
5.4. Decentralized Identifiers

As a future work the ACE-OAuth framework could be extended by Decentralized Identifiers (DIDs) [6]. Most unique identifiers that are used by persons on the Internet are in the control of third parties including email addresses or phone numbers. Therefore, these identifiers and their revocation are not under the control of the people that are using them.
The DID specification defines an identifier type that can be generated by individuals using cryptographic proofs and systems they trust in.

Decentralized Identifiers could be a meaningful extension to decentralized trust models in order to gain more independence on identities and authorization on the Internet.

6 Conclusion

In this thesis a new approach for authentication and authorization in the field of the Internet of Things was presented and an implementation based on the ACE-OAuth framework was given to verify this approach. For this purpose, the building blocks of the ACE-OAuth framework which address the limited resources in the field of the Internet of Things were introduced.
Then, the exchanged CoAP messages and computations which enable a *Client* to receive a protected resource from a *Resource Server* over a secured channel were presented. For the monitoring and pre-establishing of authorization a trust model that is based on pretty good privacy was introduced. Then, the required steps to model the trust model using a smart contract and to deploy the smart contract on an Ethereum blockchain were presented. Additionally, the communication between the *Authorization Server* and the *Resource Owner* of the ACE-OAuth framework and the smart contract were explained. Thereupon, the pre-establishing of authorization on the smart contract by the *Resource Owner* was presented as well as the transactions made by the *Authorization Server* in order to query pre-established authorization on the blockchain.
The decentralization of the trust model was realized using a smart contract that was deployed to an Ethereum blockchain.
Finally, the practicability of scenarios using "Decentralized trust models for the Internet of Things" was verified by calculating the costs of the necessary transactions and by assessing the feasibility of the decentralization of trust models.
It could be shown that "Decentralized trust models for the Internet of Things" represent a practicable alternative to centralized trust models.



License of the Documentation

Copyright (c) 2020 Flurin Trübner.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The GNU Free Documentation Licence can be read from [11].

References

[1] A. Busygin A. Konoplev and D. Zegzhda. A blockchain decentralized public key infrastructure model. *Automatic Control and Computer Science*, 52(8):1017–1021, 2018. 34

[2] K. Loupos A. Papageorgiou, A. Mygiakis and T. Krousarlis. Dpki: A blockchain-based decentralized public key infrastructure system. In *2020 Global Internet of Things Summit (GIoTS)*, pages 1–5, 2020. 3

[3] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049, October 2013. 14, 15, 17

[4] V. Buterin J. Callas D. Dorje C. Lundkvist P. Kravchenko J. Nelson D. Reed M. Sabadello G. Slepak N. Thorp C. Allen, A. Brock and H. Wood. Decentralized Public Key Infrastructure. Technical report, March 2020. 33

[5] Ed. D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. 6, 7, 8

[6] D. Longley C. Allen R. Grant M.Sbadello D. Reed, M. Sporny and J. Holt. Decentralized Identifiers. Internet-Draft, July 2020. 69

[7] W. Diffie and M. Hellman. New directions in cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, IT-22(6):644–654, November. 31

[8] J. R. Douceur. The sybil attack. In F. Kaashoek P. Druschel and A. Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. 34

[9] M. Pias R. Sharma E. K. Lua, J. Crowcroft and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys Tutorials*, 7(2):72–93, 2005. 37

[10] L. Wang F. Xia, L. T. Yang and A. Vinel. Internet of things. *JOURNAL OF COMMUNICATION SYSTEMS*, (25):1101–1102, 2012. 2

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

[12] F. Palombini G. Selander, J. Mattsson and L. Seitz. Object Security for Constrained RESTful Environments (OSCORE). RFC 8613, July 2019. 20, 21

[13] R. Greene and M. N. Johnstone. An investigation into a denial of service attack on an ethereum network. 2018. 37

[14] V. Akimenko V. Niculichev I. Zikratov, A. Kuzmin and L. Yalansky. Ensuring data integrity using blockchain technology. pages 534–539, 04 2017. 37

[15] E. Karatsiolis J. Buchmann and A. Wiesmaier. *Introduction to Public Key Infrastructures*. Springer, 2013. 36

[16] M. Furuhed J. Höglund, S. Lindemer and S. Raza. Pki4iot: Towards public key infrastructure for the internet of things. February 2020. 3

[17] E. Wahlstroem S. Erdtman L. Seitz, G. Selander and H. Tschofenig. Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth). Internet-Draft 35, June 2020. 3, 28, 42

[18] L. Lunesu. A Tale of Two Curves, November 2016. 31

[19] B. Kim M. Burhan, R. Rehman and B. Khan. Iot elements, layered architectures and security issues: A comprehensive survey. August 2019. 2

[20] G. Selander S. Erdtman M. Jones, L. Seitz and H. Tschofenig. Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs). RFC 8747, March 2020. 27

[21] S. Erdtman M. Jones, E. Wahlstroem and H. Tschofenig. CBOR Web Token (CWT). RFC 8392, May 2018. 25, 26

[22] P. Bellot M. T. Hammi, Badis Hammi and A. Serhrouchni. Bubbles of trust: A decentralized blockchain-based authentication system for iot. *Comput. Secur.*, 78:126–142, 2018. 33, 34, 35

[23] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008. 20

[24] K. Ormiston and M. Eloff . Denial-of-service distributed denial-of-service on the internet. pages 1–14, 01 2006. 34

[25] L. Bhaskari P. Babu and CH. Satyanarayana. A comprehensive analysis of spoofing. *International Journal of Advanced Computer Sciences and Applications*, 2011. 34

[26] R. Ramaswamy S. Madakam and S. Tripathi. Internet of things (iot): A literature review. *Journal of Computer and Communications*, (3):164–173, 2015. 2

[27] J. Schaad. CBOR Object Signing and Encryption (COSE). RFC 8152, July 2017. 17, 18

[28] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, August 2007. 2

[29] A. K. Singh and A. K. Misra. Analysis of cryptographically replay attacks and its mitigation mechanism. In P. S. Avadhani S. C. Satapathy and A. Abraham, editors, *Proceedings of the International Conference on Information Systems Design and Intelligent Applications 2012 (INDIA 2012) held in Visakhapatnam, India, January 2012*, pages 787–794, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 34

[30] M. Dakhilalian M. Jadliwala T. Rajab, M. Manshaei and M. Rahman. On the feasibility of sybil attacks in shard-based permissionless blockchains. *ArXiv*, abs/2002.06531, 2020. 37

[31] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F. Wang. An overview of smart contract: Architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113, 2018. 38

[32] K. Hartke Z. Shelby and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014. 10, 11, 12, 13

Referenced Web Resources

[33] Cbor jump table. https://tools.ietf.org/html/rfc7049appendix-B. 17

[34] Encryption and digital signatures using gpg. https://mran.microsoft.com/snapshot/2016-12-19/web/packages/gpg/vignettes/intro.html. 35

[35] Eth gas station. https://ethgasstation.info. 65

[36] Eth gas station calculator. https://ethgasstation.info/calculatorTxV.php. 66

[37] Ethereum opcodes. https://ethervm.io/opcodes. 38

[38] Solidity ec signature verficiation. https://github.com/tdrerup/elliptic-curve-solidity. 56

[39] Website of ganache. https://www.trufflesuite.com/ganache. 47

[40] Website of implementation. https://github.com/FlurinT/MT. 47

[41] Website of the npm module: cbor. https://www.npmjs.com/package/cbor. 45

[42] Website of the npm module: coap. https://www.npmjs.com/package/coap. 45

[43] Website of the npm module: coap-router. https://www.npmjs.com/package/coap-router. 45

[44] Website of the npm module: cose-js. https://www.npmjs.com/package/cose-js. 45

[45] Website of the npm module: crypto. https://nodejs.org/api/crypto.html. 46

[46] Website of the npm module: ec-pem. https://www.npmjs.com/package/ec-pem. 46

[47] Website of the npm module ethereumjs-util. https://www.npmjs.com/package/ethereumjs-util. 47

[48] Website of the npm module: futoin-hkdf. https://www.npmjs.com/package/futoin-hkdf. 46

[49] Website of the npm module: web3.js. https://github.com/ethereum/web3.js/. 46

[50] Website of the sodility documentation. https://solidity.readthedocs.io/en/v0.7.0/. 47

[51] Website of the trufflesuite. https://www.trufflesuite.com/. 47

List of Figures

2.1. OAuth2 Authentication Flow 8
2.2. CoAP Layers 10
2.3. Reliable Messaging 11
2.4. CoAP Message Format 11
2.5. CoAP Request Messages 12
2.6. CoAP Response Messages 13
2.7. COSE Message Types 18
2.8. CoAP Layering using OSCORE 21
2.9. Matching of Contexts 21
2.10. Defined Claims 26
2.11. PKI compared to the Web of Trust, from [34] 35
2.12. A Public Key Signing Meeting 36
2.13. Single Key Ring Entry 36

3.1. Workflow of the Implementation 42

4.1. Recommended Gas Prices for Processing Times 65
4.2. Gas Prices related to the Acceptance of Transactions 66

Listings

2.1. CoAP/S URI Scheme 14
2.2. CBOR Array Encoding 17
2.3. COSE-Sign1 Structure 19
2.4. COSE-Signature Object 19
2.5. COSE Sig Structure 19
2.6. Info Object Parameter 23
2.7. COSE-Key in a PoP Token cnf Claim 27
2.8. Example CWT Token 27

3.1. CoAP Server Instance 45
3.2. Endpoint using coap-router 45
3.3. CBOR Encoding and Decoding 45
3.4. COSE Sign and Encrypt 46
3.5. HKDF for OSCORE 46
3.6. EC Key Generation and ECDH 46
3.7. PEM Key Formatting 46
3.8. Smart Contract Parameter Encoding 46
3.9. Matching Buffer Format 47
3.10. Client Constructor 48
3.11. CoAP Token Request 48
3.12. Authz-info token upload 48
3.13. ES256 COSE Signing 49
3.14. ES256 COSE Sign Verification 49
3.15. Token Request Endpoint 50
3.16. Signature Verification 51
3.17. Access Verification 51
3.18. Access Token Response 52
3.19. Claim Key Translation 53
3.20. Claim Key Registry 53
3.21. Key Translation 54
3.22. Security Context Constructor 55
3.23. Deriving the Security Context 55
3.24. Derivation of the Sender Key 55
3.25. Public Key and Revocation registry 56
3.26. Signature Validation 56

3.27. Adding a Public Key to a Key Ring . . . 56
3.28. Key Revocation . . . 57
3.29. Key Ring . . . 57
3.30. Key Ring Creation . . . 57
3.31. Access . . . 58
3.32. Provisioning Access . . . 58
3.33. Contract deployment . . . 59
3.34. Smart Contract function wrapper . . . 59
3.35. authz-info Endpoint . . . 60

4.1. CBOR Formatted Token Request Claims . . . 62
4.2. Token Request in Diagnostic Notation . . . 62
4.3. Pre-established Access returned from the Smart Contract . . . 63
4.4. Payload of the COSE Sign Token Response . . . 63
4.5. POP Access Token . . . 63
4.6. Resource Response . . . 64