Constrained IoT authentication in Cloud services

Tomás Silva
Instituto Superior Técnico
Truphone
Lisbon, Portugal
tomas.silva@truphone.com

***Abstract*—Constrained devices operating on constrained networks are a big part of the IoT paradigm. They have a myriad of applications such as farming [1], wheater-monitoring [2] and logistics and supply-chain management [3]. With Cloud Services and IoT technologies on the rise, it is imperative to create a seamless authentication mechanism that can identify these devices to a third-party Cloud provider. A lightweight solution to this problem is explored, by leveraging network-level authentication via SIM to identify and authenticate these constrained devices operating on constrained networks to a third-party Cloud provider. This work presents an authentication protocol that is transparent to the client, specifically designed for constrained devices operating on constrained networks. This is achieved using an OAuth 2.0-based protocol running on top of improved communication and application-layer protocols, with the introduction of a new key feature. In this novel approach, the broker component allows the client to be free from all authentication procedures and allows the delegation of all of these tasks, like managing and requesting access tokens. When compared to the authentication mechanisms currently working in Truphone, the proposed solution reduces the client-side network traffic by more than 98%, reduces computation needs and is suitable to use over NB-IoT, the cellular network that will power these constrained devices.**

I. Introduction

Most authentication mechanisms that allow authentication for third-party services are not optimized for the IoT paradigm, targeting regular computers, a radically different scenario from the one that will be explored. On the other hand, most authentication mechanisms designed specifically for IoT are made with an entirely different objective, double way authentication between two IoT devices, and not authenticating the IoT device to a third-party service.

This work leverages the existent network-based authentication to trustfully and automatically authenticate a device to a third-party service, such as a Cloud provider (e.g. AWS IoT Core). Since the network provider can already authenticate and trust a device, by extending this trust with a mechanism to authenticate the device to a third-party, the same trustful and automatic authentication can be achieved. Constrained devices operating on constrained networks currently do not have out-of-the-box solutions for lightweight authentication to a third-party. Since constrained devices and constrained networks play an important role in a large scope of IoT (Internet of Things) scenarios, this is an industry shortcoming that this work attempts to solve.

II. Background

A. OAuth2.0

OAuth 2.0 [4] is an authorization framework that allows users to authenticate towards a remote resource/service. This framework defines four participants: the client, the resource owner, the resource server and the authorization server. The **client** is the entity that requests access to a protected resource on behalf of the resource owner, and it can only access the protected resource once it has been granted authorization. The **resource owner** is the entity that can grant permission to the client to access said resource. The **resource server** is the entity hosting the protected resource, allowing access to the resource when presented with an access token. The **authorization server** is the entity capable of issuing access tokens after successfully authenticating the resource owner and obtaining permission.

1) Protocol Flow: The interactions between these roles are depicted in Figure 1.

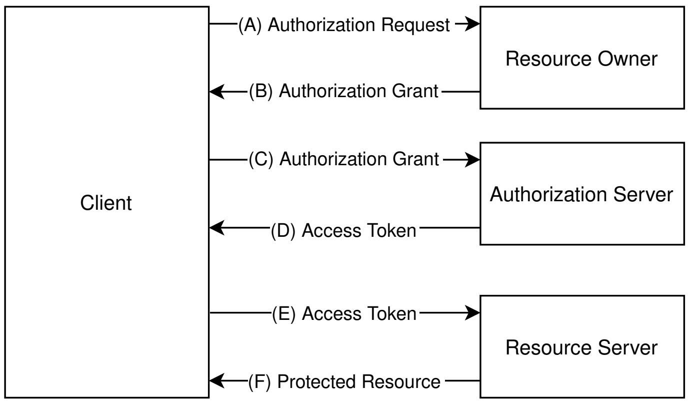

Fig. 1. OAuth 2.0 Protocol

(A) marks the start of the interaction between all parties, with a request for authorization to access the protected resource, made by the client to the resource owner. **(B)** depicts the response to the authorization request, where the resource owner grants the client authorization to access the protected resource. In **(C)**, the client presents this authorization grant to the authorization server which, after authenticating the client and validating the authorization grant, issues an access token that is sent to the client in **(D)**. In **(E)**, the client requests the protected resource to the resource server, sending the access token in the request. The resource server validates the access token and responds to the client by allowing access to the protected resource in **(F)**.

An important caveat is that the client works on behalf of the resource owner, meaning that steps **(A)** and **(B)** are usually not actual message exchanges between different entities but the a end-user, acting as the resource owner, using an application, acting as the client, to access information that he controls on another service/application.

2) Confidential Clients and Public Clients: OAuth 2.0 makes an important distinction regarding clients. Clients can either be public or confidential. **Confidential** clients can hold credentials in a secure way without exposing them to unauthorized parties. **Public** clients cannot. So confidential clients are, for example, web applications running in a secure server and public clients are, for example, single-page applications running directly on a browser.

3) Resource Owner Password Credentials Flow: The resource owner password credentials flow is designed for cases when the resource owner has a trust relationship with the client. In this flow, the client has access to the resource owner credentials. This flow is only used when other flows are not viable and it can be used to migrate clients to OAuth 2.0 by converting the stored resource owner credentials into access tokens. This flow is depicted in Figure 2.

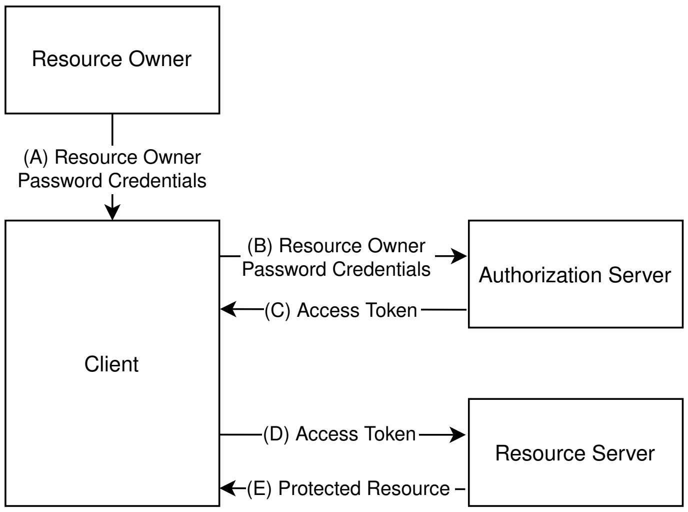

Fig. 2. Resource Owner Password Credentials Flow

The protocol starts in **(A)**, here the resource owner provides the client with its credentials. The way these credentials are shared is out of the scope of the protocol, so, as long as the client can access the resource owner's credential, step **(A)** is completed. In **(B)** the client requests an access token from the authorization server and sends the resource owner credentials along with this request. After validating the resource owner's credentials, the authorization server issues an access token that is sent to the client in **(C)**. In **(D)**, the client uses this access token to request access to a protected resource held by the resource server. After validating the access token, the resource server grants access to the protected resource, depicted in the Figure as **(E)**.

4) Client Credentials Flow: The client credentials flow is used when the client is requesting access to resources owned by itself or when it is accessing resources under the control of another resource owner. This resource owner has previously orchestrated, with the authorization server, that the client can access it without contacting the resource owner (the way this orchestration is performed is not defined by OAuth 2.0 and it is considered out of scope of the standard). In these cases, there is no need to contact the resource owner and the client can access the protected resource by authenticating itself to the authorization server. In this flow, the client directly contacts the authorization server and requests the access token. This flow is illustrated in Figure 3.

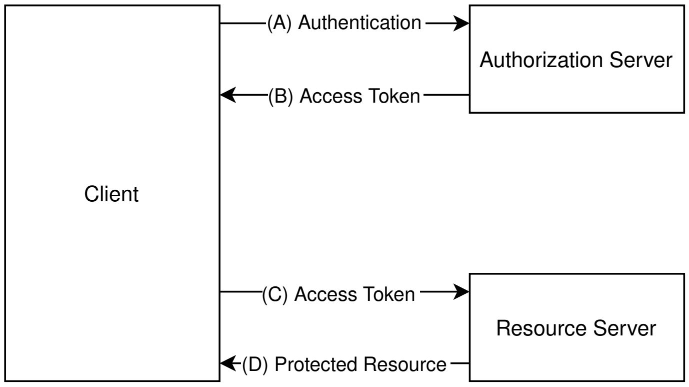

Fig. 3. Client Credentials Flow

In **(A)** we can see that the client authenticates itself to the authorization server (this authentication can be done by any method; a common one is to use a username and a password to identify a client). The authorization server, after authenticating the client, issues an access token which is represented in **(B)**. **(C)** and **(D)** are the same steps shown in Figure 1 where the client sends the access token to the resource server and the resource server allows the client to access the protected resource.

5) Access Token: The access token is the token that proves that the client has the authorization to access the protected resource. This access token has the form of a JWT. The access token to be considered valid must be a valid JWT and the resource server must be able to validate this token. This validation includes validating the signature and the "aud" claim which must indicate that the token was issued to access the determined protected resource.

B. ACE Framework

The Authentication and Authorization for Constrained Environments (ACE) Framework [5] is a framework designed to expand the authorization flow of OAuth 2.0 to constrained devices in IoT networks. One key difference that aims to accommodate the limitations of constrained devices and networks is the use of CWTs instead of JWTs. By using CBOR, the protocol can better compress the messages, resulting in smaller payloads.
The ACE protocol, illustrated in Figure 4, works similarly to OAuth 2.0. It starts when the client (the constrained device) asks the Authorization Server for an access token that contains claims about the identity of the client and if the client can access the protected resource here depicted in **(1)** and **(2)**. Then, the client sends this access token in the request to access the resource hosted by the Resource Server, as shown in **(3)**. The Resource Server, if needed, can verify the token with

the Authorization Server using an introspection request. This introspection request is portrayed in steps **(4)** and **(5)**. The resource server sends the Access Token, that the client used, to the Authorization Server. The Authorization Server then confirms the validity of the Access Token. After validating the token (with or without an introspection), the Resource Server provides the client with access to the protected resource, as shown in **(6)**. This protocol was made to work over CoAP however, HTTP implementations are possible.

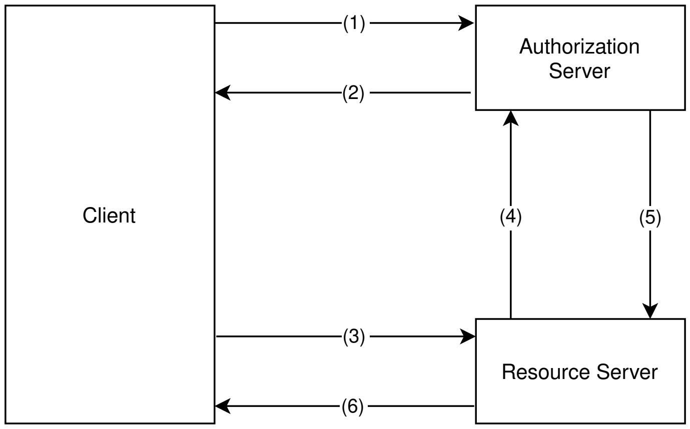

Fig. 4. ACE Protocol Flow

The implementation in [6] is made over HTTP due to the poor support of DTLS by the networking libraries implementing CoAP servers and clients showed poor support for DTLS at the time. This implementation showed that, by using CWT instead of JWT, the access tokens were about 50% smaller. The reported overhead due to the use of HTTP is around 50%, which, when dealing with constrained devices and networks, is a very large number. This implementation is extremely useful in showing the massive difference in performance that is possible by using CWT instead of JWT.
This protocol was made to work with two different security protocols, one working with Object Security for Constrained RESTful Environments (OSCORE) [7] and another working only with DTLS. OSCORE allows for true end-to-end encryption (over proxies) while DTLS does not, since CoAP defines a number of proxy operations that require transport layer security to be terminated at the proxy. It is important to notice that OSCORE runs on the application layer and DTLS on the transport layer so they are not mutually exclusive.

C. Truphone's Existing Solution

Truphone has a patent pending working solution (patent ID PCT/GB/2021/051093) capable of authenticating IoT devices equipped with a Truphone SIM to a third-party. It is based on the Client Credentials Flow from OAuth2.0, described above. This solution, also known as the "notarizer", is here described starting with the participants in the protocol and then detailing the flow.

1) Truphone's Private Network: This solution leverages the authentication and security present in cellular networks. An eSIM equipped device, using LTE/NB-IoT, has a secure channel to the core cellular network. The device is connected to the core network which then connects to the Internet. This core network/secure channel will be named Truphone's Private Network. The authorization server in Truphone's existing solution is deployed in this core network, where it has access to a mapping between a certain client's IP and their SIM credentials. This is caused by the authentication process in cellular networks. A client that has access to the Internet through a cellular network is authenticated beforehand via SIM. The cellular provider is responsible for IP address provisioning and it can associate this IP to a certain SIM card. By leveraging this association, the authorization server knows the identity of every client that is connected to Truphone's network.

2) Protocol Participants: As mentioned in the previous section, the "notarizer" is based on OAuth 2.0 and so they share some participants, namely the authorization server and the client. The client, as in OAuth 2.0, is the participant that needs to be authenticated, the authorization server is the entity that has the ability to identify and authenticate the client and, in OAuth 2.0, there is the resource owner who needs to assess the identity of the client and in the "notarizer" this role is filled by the third-party, which needs to assess the identity of the client. These participants will now be detailed.

Authorization Server: The authorization server has the same responsibilities as its namesake in the ACE framework and OAuth 2.0. It is responsible for identifying clients and issuing access tokens that assert a client's identity. This authorization server however has a major difference from the ones presented in the aforementioned protocols. It is capable of authenticating every client that issues a request without the need for credentials to be exchanged. The authorization server is a service provided by Truphone and it is deployed in Truphone's network. This means that the authorization server has a secure channel established to the client and can identify it.

Client: The client is the entity that needs to be authenticated to another entity by the authorization server (in this solution to the third-party service). In this proposed solution, the client is a constrained IoT device operating in a constrained environment, with a Truphone SIM. The client also has a third-party Software Development Kit (SDK) pre-installed which provides the client with the necessary information to contact the third-party service, namely the IP address. The client is connected to the Internet using a Truphone eSIM. This connection allows the client to communicate with Truphone's services using a private network.

Third-Party Service: The third-party service is the equivalent of the resource owner in the ACE framework and OAuth 2.0. It expects clients to request access to some protected resource and it relies on the authorization server for the identification of these clients. Working with the authorization server can provide, or deny, access to this resource. In this solution, due to third-party constraints, the client uses MQTTS (with one-way authentication, so the third-party authenticates itself to the client, which has a preloaded certificate to confirm the third-party identity) to communicate with the third-party.

3) Protocol Flow: Truphone's existing solution's flow is depicted in Figure 5.
In **(1)**, the client, through an HTTP request, asks for a token to the authorization server. In this request, the client will send to the authorization server SIM credentials (namely two unique

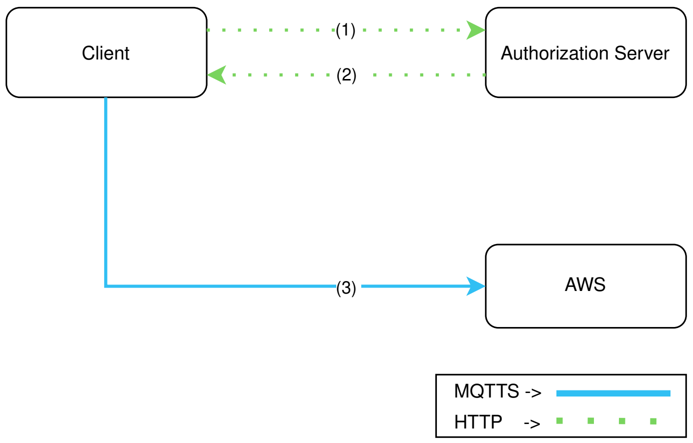

Fig. 5. Notarizer

identifiers: ICC-ID and IMSI). Since the authorization server is deployed in Truphone's private network, it has access to the mapping between a certain client's IP address and their SIM credentials, allowing the authorization server to identify the client. In **(2)**, the authorization server responds to this request with the token issued for the client, and finally, in **(3)**, the client uses this token to authenticate itself towards the third-party service using MQTTS. An important detail, is that while the connection between the Client and the Authorization Server is not protected by HTTPS, it is a private connection, running on Truphone's private network. This flow is triggered when the client needs to publish a message to a third-party.

4) ID Token: The token follows the specification defined in the OAuth 2.0 standard. It is a JWT containing the mandatory OAuth 2.0 claims and signed by the authorization server. Token validation is done by the third-party. Token validation also follows the OAuth 2.0 standard. A key-sharing mechanism between the authorization server and the third-party is not defined as it is considered out-of-scope for this project.

5) Limitations and Advantages: The "notarizer" is not the best solution when dealing with constrained devices for a myriad of reasons. Firstly, the use of HTTP is a problem. HTTP runs on top of TCP which, provides a connection-oriented reliable service at the cost of additional overhead when compared with UDP. The use of a JWT poses another problem. There are other alternatives that can encode the same information as a JWT while saving space, namely a CWT which trades off human readability for compactness. Another problem is the use of MQTTS. MQTTS is a version of MQTT which runs on top of TLS. The use of MQTT by itself could already be considered problematic considering that it runs on top of TCP, however, the use of TLS makes this even worse. The TLS handshake presents an unacceptable overhead when dealing with constrained devices.

6) Security: To prove the security of Truphone's existing solution, an assumption was made: OAuth 2.0 is a secure protocol, designed for use on the Internet over HTTPS. The security of the intermediate solutions will be demonstrated by addressing the changes it has compared to OAuth 2.0.
The "notarizer" implements the following changes compared to OAuth 2.0:

1) The use of HTTP instead of HTTPS, when requesting the token
2) The use of MQTTS instead of HTTPS, when sending the token to authenticate itself

Both are changes to application-layer protocols. The second change is meaningless as HTTPS security comes from the use of TLS and MQTTS also runs over TLS. This means that both HTTPS and MQTTS provide the same security capabilities. The first change, however, can pose problems as HTTP is not secured by TLS. Nonetheless, this connection between the client and the authorization server is, as mentioned before, made over Truphone's Private Network that guarantees authentication and confidentiality, keeping the protocol secure.

III. Proposed Solution

A. Problem Description

This work aims to solve the lack of an automatic authentication mechanism for constrained IoT devices operating in constrained environments that allows devices to authenticate themselves towards third-party services. The proposed solution targets constrained devices that, due to operating in constrained networks, cannot use the current solution provided by Truphone due to network overhead. The proposed solution is based on the current working solution provided by Truphone and it is designed specifically for constrained devices.

B. Building Blocks

One of the key blocks of the proposed solution is **NB-IoT**. NB-IoT is the ideal network when dealing with constrained devices and environments. This network is here considered to connect the devices to the Internet.
The second building block is **CoAP**. CoAP is built specifically to cope with constrained devices and environments. This solution uses non-confirmable messages as the acknowledgement messages introduced by the reliability mechanism would originate unnecessary overhead to the protocol. The lightweight nature of this protocol, when compared to other application-layer protocols, makes it the best alternative to use. It is used when the devices need to send or receive data from other entities.
The third and final building block is **CBOR** and, more specifically, **CWT**. The use of CWTs will allow the creation of a more compact access token.

C. Intermediate Solution 1

The first intermediate solution is an adaptation of the existing solution. It uses the same protocol participants as the "notarizer". In order to reduce traffic network on the client side this first approach uses different data transfer protocols and a different method to encode the token. This solution presents four major changes:

1) It uses CoAP instead of using HTTP when the client requests the token from the authorization server
2) It uses a CWT instead of a JWT to encode the access token
3) Token validation is now performed by the authorization server

4) The client no longer sends SIM data to the authorization server. The authorization server can associate an IP to SIM data so the client's IP is sufficient for the authorization server to identify a client, since they share Truphone's private network.

The first two changes will reduce the client's traffic network. By using a lighter application-layer protocol and a more efficient data format for the access token, the client will transmit fewer bytes when it requests the token, when it receives it and when it publishes the message in the third-party as the token is present in that message. The third change tackles the need to manually distribute the authorization server's public key as the token validity is now confirmed by the authorization server. The access token is equal to the one used in the current solution apart from the different data formats. The last change will allow the token request to be slightly smaller since there is no need to send any payload.

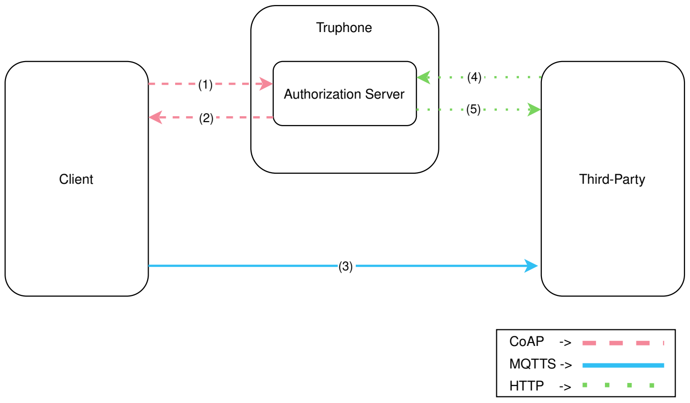

Fig. 6. Intermediate Solution 1 Flow

1) Protocol Flow: The protocol flow is represented in Figure 6. In **(1)** and **(2)**, the client requests the token to the authorization server. These two messages are similar to the ones made in the "notarizer" flow with the already mentioned changes of using CoAP instead of HTTP and the use of a CWT instead of a JWT to encode the token. Message **(3)** is the same as in the "notarizer" sending both the token and the message to publish to the third-party using MQTTS. Messages **(4)** and **(5)** are introduced here. In **(4)** the third-party sends the access token received in **(3)** to the authorization server and the authorization server will assess the validity of the token and send it back to the third-party in **(5)**. This flow is triggered when the client needs to publish a message to a third-party.

2) Limitations and Advantages: While this intermediate solution solves some of the "notarizer" limitations, like the need to manually distribute the authorization server's public key and the use of HTTP, it still has a problem: the use of MQTTS is still an issue that brings unacceptable communication overhead for a constrained device to use, since it runs on TCP and it uses TLS to secure communications.

3) Security: To prove the security of intermediate solution 1, the changes it introduces from Truphone's existing solution will be analyzed.

Intermediate solution 1 implements the following changes compared to the "notarizer":

1) It uses CoAP instead of using HTTP when requesting the token
2) It uses a CWT instead of a JWT to encode the access token
3) Token validation is now performed by the authorization server
4) The client no longer sends SIM data to the authorization server. The authorization server can associate an IP to SIM data so the client's IP is sufficient for the authorization server to identify a client, since they share Truphone's private network.

The first change is inconsequential as the security comes from the private network, not from the application-layer protocol chosen here. The second change also does not affect security, since the CWT is properly signed as it was before. The third change is not a problem as token validation can be done by any party that possesses the public key from the key pair used to sign the access token. As long as the connection between the third-party and the authorization server is secure and the third-party trusts the authorization server, this is secure. The fourth change is meaningless to security, and as long as the client can send some information that the authorization server can use to identify and authenticate the client, this is not a problem.

D. Intermediate Solution 2

To improve the above solution, in particular the use of MQTTS. This intermediate solution introduces a new protocol participant, the broker. The broker connects the client to the third-party service, as illustrated in Figure 7. It is introduced to allow the client to use CoAP even if the third-party is not prepared to use it. It acts as a middle-man between the client and the third-party service, receiving CoAP requests from the client and extracting the information meant to the third-party service crafting, from this, an appropriate request is sent to the third-party. The type of request depends on the third-party. In the case of the current solution, the third-party, as mentioned before, uses MQTTS. In this case, the broker receives a CoAP request, extracts its information and uses MQTTS to send this information the third-party on behalf of the constrained device. The addition of the broker is a crucial step as it enables the use of constrained devices operating in constrained environments by allowing the device to communicate exclusively using CoAP, which, as seen before, is the best fit for this kind of device. The broker is also deployed in Truphone's private network and so, it has access to the same features as the authorization server mentioned above, namely a private confidential connection to the client. The access token and its validation process is the same to the one used in intermediate solution 1.

1) Protocol Flow: The protocol flow, depicted in Figure 7. Messages **(1)** and **(2)** stay the same as in the first intermediate solution. Message **(3)**, as in the first intermediate solution, sends the access token and the message, however, in this case, the client does not send this directly to the third-party. It sends it to the broker which will then send this information to the third-party on behalf of the client. It also uses CoAP instead of MQTTS. Since MQTTS has a reliability mechanism and CoAP does not, the broker sends the message **(4)** to confirm that it received the message, and when paired with a timeout it can be used to create an application-layer reliability mechanism. In message **(5)** the broker sends the access token and the message to the third-party using MQTTS as the third-party dictates. Messages **(6)** and **(7)** perform token validation. The third-party sends the access token to the authorization server, that responds with the assessment of the token's validity. This token validation process is the same as in the first intermediate solution and stay the same. This flow is triggered when the client needs to publish a message to a third-party.

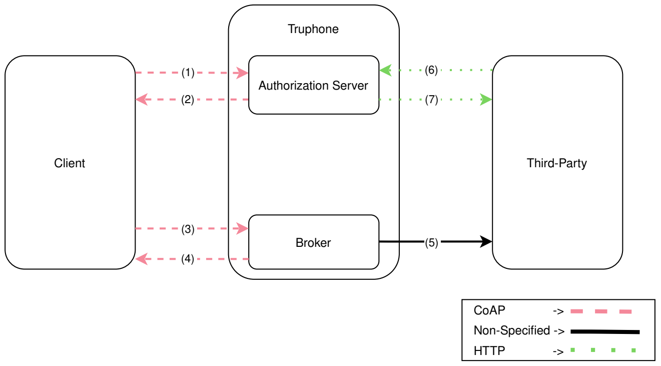

Fig. 7. Intermediate Solution 2 Flow

2) Limitations and Advantages: This solution does not have any major limitations like the ones presented before. The use of CoAP instead of MQTTS requires the aforementioned application layer reliability mechanism to be implemented. The broker introduces a layer of abstraction that allows a client to send a message to a third-party in a transparent way. One use case where the broker can be useful is in managing the client. Since the client is expected to be a constrained device, it is reasonable to assume that there will be some entity managing it. In the case of migrating a client or a collection of clients from a third-party to another (for example from AWS to another cloud provider), this can be done without changes in the client since the broker will deliver these messages, and just needs to be informed of the change so it can behave accordingly.

3) Security: To prove the security of intermediate solution 2, the changes it introduces from intermediate solution 1 will be analyzed.
Intermediate solution 2 implements the following changes compared to intermediate solution 2:

1) The introduction of the broker
2) The use of CoAP instead of MQTTS by the client

Starting with the first change. The broker is deployed inside Truphone's private network and so its connection to the client benefits from the same guarantees as the connection between the client and the authorization server. The connection between the broker and the third-party is made via MQTTS, which, as seen before, is secure.

E. Final Proposed Solution

The proposed solution optimizes the second intermediate solution by delegating the responsibility of requesting a token from the client to the broker. This approach allow for a reduction of the traffic network on the client side since messages **(1)** and **(2)** from the previous solution will not be present in this solution, as illustrated in Figure 8. In this approach the token request is made by the broker.

1) Protocol Flow: The protocol flow is represented in Figure 8. In message **(1)**, the client sends to the broker the data it needs to publish in the third-party. The broker confirms the reception of this message with **(2)** for reliability. The broker will then extract the client's IP and send it in **(3)** a token request to the authorization server which the authorization server can identify, as mentioned before. In **(4)**, the authorization server sends the access token identifying the client to the broker which will then compose a message using this token and the message sent by the client in **(1)** to the third-party, on behalf of the client, in **(5)**. Messages **(6)** and **(7)**, where the third-party requests the authorization server to assess the validity of the access token, will stay the same, as in the previous solutions.

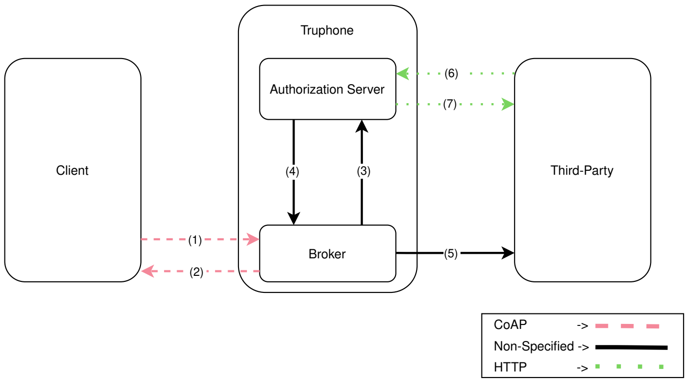

Fig. 8. Proposed Protocol Flow

2) Limitations and Advantages: By delegating the token request to the broker, this protocol essentially builds a transparent authentication scheme for the client. The client only needs to send to the broker the message that it wants to publish to the third-party and the broker will deal with everything from here. By doing this, the solution optimizes traffic network on the client-side as it only needs to send the message that it would need to send anyway and it can do it in a near-optimal way by using CoAP instead of other heavier protocols. It also allows for client managing like in intermediate solution 2.

3) Security: To prove the security of the final proposed solution, the changes it introduces from intermediate solution 2 will be analyzed.
The final proposed solution implements the following changes compared to intermediate solution 2:

1) The delegation of the token request from the client to the broker

This change does not affect security since the connection between the authorization server and the broker is made on Truphone's private network. The broker uses the client's IP to identify it to the authorization server like before and so there is no change to the protocol's security, proving this solution to be secure.

F. Access token

The access token, is a CWT. As aforementioned, a CWT must be either signed, MACed or encrypted. The access token

must preserve its integrity and it must be verifiable that it was issued by the Authorization Server. Given these requirements, the access token will be signed by the Authorization Server. This presents another choice that must be made: should it be signed using an ECDSA or an EdDSA algorithm. According to [8], the curve P-256 should be used to generate keys used for digital signatures for authentication purposes. A P-256 curve generates a 64-byte key. As per [9], ECDSA signatures are 2 times longer than the signer's private key for the curve used during the signing process. This means that using ECDSA would generate a 128 byte sized signature. According to [10], EdDSA can use two curves for key generation: the edwards25519 and edwards448 curves. EdDSA public keys have exactly b bits and EdDSA signatures have exactly 2b bits. The value b is a multiple of 8, therefore, the public key and signature lengths are an integral number of octets. For Ed25519, b is 256, so the private key is 32 octets. With a private key of 32 bytes, EdDSA using the edwards25519 curve would generate a signature with 64 bytes. Since both algorithms are secure [10], the access token is signed using EdDSA with the edwards25519 curve, also known as an Ed25519 digital signature, to produce a smaller signature.
Apart from the signature, the access token contains claims about the subject that possess it. Unlike a JWT, a CWT does not have any mandatory claims that must be made about a subject. The access token provided by the current solution contains the claims mandatory as per the OAuth 2.0 standard. This includes the issuer, subject, expiration date and audience claims. The access token in the proposed solution contains the same claims. The claim "subject" needs to uniquely identify a subject so the authorization server uses the ICC-ID and the IMSI of the client in this claim in the following way: "ICC-ID"|"IMSI". This value will allow any client to be uniquely identified.

Validating an Access Token: The tokens are validated by the authorization server. The validation process of an access token has 4 steps:

1) It is necessary to validate the Ed25519 digital signature, assuring that it was signed with a Truphone's private key
2) It is necessary to confirm that the issuer claims match "Truphone"
3) It is necessary to confirm if the access token is not expired
4) It necessary to confirm if the audience claim matches the identity of who wants to validate the token

If any of these steps fail, the token is considered invalid.

G. Improvements

The proposed solution was designed to solve the current's solution shortcomings, namely:

1) The use of HTTP
2) The use of JWT
3) The use of MQTTS
4) Lack of third-party interoperability

To solve issue 1., the proposed solution uses CoAP. To solve issue 2., the proposed solution uses CWT encoded tokens. Issue 3. is directly related to issue 4. as it originates from a third-party restriction. To deal with issues 3. and 4., the broker is introduced in the proposed solution. It allows the client to use CoAP for every communication and it eliminates the hassle of modifying multiple clients to deal with different third-parties. It delegates this effort to the broker making this process easier as one broker will serve multiple devices.

H. Implementation Details

1) Third-party service: The third-party used in all implementations (including the baseline) is AWS IoT Core. AWS IoT Core is a cloud service from AWS made for IoT devices. The use of this third-party shapes this implementation as AWS IoT Core requires connecting devices to use either HTTP, MQTT or LoRaWAN. Keeping in mind the objective to minimize traffic network, the connection to AWS IoT Core was made in MQTT. MQTT was made specifically for constrained devices and it is lighter than HTTP. LoRaWAN is a low power wide area networking protocol made for IoT connectivity [11], however, it requires additional infrastructure, namely LoRaWAN gateways and network servers. This brings additional constraints to the client which would need to set up this infrastructure, making this option a less flexible alternative than using CoAP. AWS IoT Core requires MQTTS to be used. This means that the connection with AWS is protected by the use of TLS version 1.2 and AWS is authenticated to the client, the messages exchanged are encrypted and their integrity is protected.

2) Authorization Server: The authorization server runs on Truphone's network and it has access to a mapping between devices' IP addresses and ICC-IDs. The authorization server is built in Rust. Rust is a modern, statically typed language that offers better memory usage than other modern and statically typed languages (Java and Go) [12]. Due to business constraints, excessive memory usage is a concern (higher cost) and Rust also supports CoAP and CBOR (by using the coap [13] and serde_cbor [14] crates). Rust however lacks support for CWT usage which dictated the need to develop a crate that added this feature. The cwt crate was developed on top of the aforementioned serde_cbor crate which offers serialization and deserialization support for CBOR objects, however, it can not create or read COSE headers that are necessary for CWTs.

3) Broker: The broker is a service built in Rust deployed on Truphone's network. It receives CoAP requests from the clients and redirects them to the third-party service using MQTTS. In this proof-of-concept, only one third-party will be used and so the broker contains the needed CA certificate to verify the third-party identity. In future use, the broker should be agnostic to the third-party.

IV. Implementation

A. Third-party service

The third-party used in all implementations (including the baseline) is AWS IoT Core. AWS IoT Core is a cloud service from AWS made for IoT devices. The use of this third-party shapes this implementation as AWS IoT Core requires connecting devices to use either HTTP, MQTT or LoRaWAN. Keeping in mind the objective to minimize traffic network, the connection to AWS IoT Core was made in MQTT. MQTT was made specifically for constrained devices and it is lighter than HTTP. LoRaWAN is a low power wide area networking protocol made for IoT connectivity [11], however, it requires additional infrastructure, namely LoRaWAN gateways and network servers. This brings additional constraints to the client which would need to set up this infrastructure, making this option a less flexible alternative than using CoAP. AWS IoT Core requires MQTTS to be used. This means that the connection with AWS is protected by the use of TLS version 1.2 and AWS is authenticated to the client, the messages exchanged are encrypted and their integrity is protected.

B. Authorization Server

The authorization server runs on Truphone's network and it has access to a mapping between devices' IP addresses and ICC-IDs. The authorization server is built in Rust. Rust is a modern, statically typed language that offers better memory usage than other modern and statically typed languages (Java and Go) [12]. Due to business constraints, excessive memory usage is a concern (higher cost) and Rust also supports CoAP and CBOR (by using the coap [13] and serde_cbor [14] crates). Rust however lacks support for CWT usage which dictated the need to develop a crate that added this feature. The cwt crate was developed on top of the aforementioned serde_cbor crate which offers serialization and deserialization support for CBOR objects, however, it can not create or read COSE headers that are necessary for CWTs.

C. Broker

The broker is a service built in Rust deployed on Truphone's network. It receives CoAP requests from the clients and redirects them to the third-party service using MQTTS. In this proof-of-concept, only one third-party will be used and so the broker contains the needed CA certificate to verify the third-party identity. In future use, the broker should be agnostic to the third-party.

V. Evaluation

A. Metrics

To test the effectiveness of the proposed solution a network traffic test is considered.
Constrained devices in constrained environments need to minimize the amount of data exchanged. By measuring the traffic in the network, it is possible to test the effectiveness of the proposed solution.
Network traffic is affected by the amount of data in the network. This data can be separated into two distinct parts: the payload and the network overhead. The payload is the content of a message. The network overhead is all the headers of lower-layer protocol data and information that is necessary for the proper transmission of a message. In the case of the proposed solution, the payload size is related to the token size and message size. By measuring token size, it is possible to understand both the improvements that this solution provides, when compared with other authentication mechanisms, and the overhead impact on the network's traffic and how the use of CoAP affects the message size when compared to other transport-layer protocols. It is also interesting to see the differences in data transmission at each OSI layer and for the data received and sent by the client, since both operations have different costs for IoT devices. Another interesting discussion is to assess if the proposed solutions are suited for NB-IoT.

B. Test set-up

To test the network traffic the following set-up was used for every tested solution ("notarizer", intermediate solutions 1 and 2 and the final solution). The client was run on a PC connected to the Internet. Wireshark was used to collect every packet sent between the client and the authorization server and between the client and the third-party (or broker in the case of the second prototype). Every test was performed 10 times. The network's conditions, depicted in Table I, were also tested, which depict the packet loss and RTT (Round-Trip Time). This was obtained by pinging the authorization server, the broker and the third-party 100 times.

TABLE I
Network Conditions

	Client to Authorization Server	Client to Third-Party	Client to Broker
Average RTT (ms)	55	74	62
Packet Loss (%)	0	0	0

By using Wireshark in the client, it is possible to test the entire traffic network on the client-side which is what is important since the client is the constrained node that needs to minimize the network traffic. It also allows the examination of the traffic per layer and to examine the data contents, allowing for an analysis of not only the overhead in each layer but also the difference in data (excluding protocol overhead) sent in each solution which corresponds to the token size and message sent. In every test, the message published by the client was a 3-byte message.

C. Experimental Results

The experimental results are presented per tested solution and are divided into two flows, in order to better assess the impact of each particular change. The first flow is the "Get Token" flow in which the client contacts the authorization server to request the token. The second flow is the "Client Publish" flow where the client sends the message, either to the broker or directly to the third-party. They are also divided by sent and received data by the client-side.
Starting by presenting the average bytes, with the standard deviation, sent and received by the client in each layer for each flow for each iteration. There are two flows considered: the flow where the client requests the access token and the flow where the client sends a message to the third-party. These flows will be named as "Get Token" flow and "Publish" flow, respectively. Then a comparison between the total bytes received and sent in each iteration, will be presented.

1) "Get Token" Flow:

Starting with the depiction of the "Get Token" flow. Table II depicts the bytes received and sent by the client. The same information, divided by OSI layer, is depicted in Figures 9 and 10

TABLE II
Bytes received and sent by the client in the "Get Token" flow for every solution

	Baseline	Solution 1	Solution 2	Solution 3
Received	969 ± 0	179 ± 0	179 ± 0	0
Sent	525 ± 0	55 ± 0	55 ± 0	0

The decrease in traffic network is clear with the changes introduced in the first intermediate solution. By making simple modifications such as changing the application-layer protocol, the data format used for the token and by authenticating clients via IP instead of SIM data, an 81.53% reduction in bytes received and an 89.52% reduction in bytes sent was observed at intermediate solution 1 (and 2, that uses the same mechanism). Solution 3 is optimal since the client does not request the token leaving this task for the broker and so sending and receiving 0 bytes.

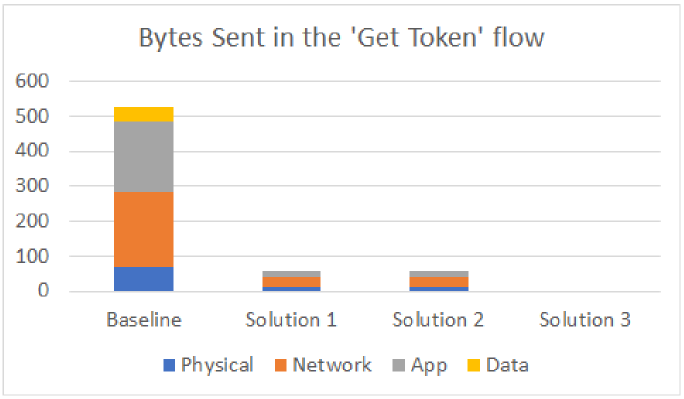

Fig. 9. Bytes Sent in the “Get Token” flow

In Figure 9 it is possible to see a significant decrease, when comparing the Baseline to the proposed solutions, in bytes sent by the client when requesting the token. This decrease is explained by the use of CoAP instead of HTTP. In the physical layer, we see a decrease in bytes sent that is related to the decrease in the number of packets sent. In the network layer, the use of UDP instead of TCP, explains this decrease. In the application layer, the decrease is caused by the reduced overhead that CoAP has when compared to HTTP. In both prototypes the CoAP request does not carry any payload as the only information needed to identify a client is their IP in Truphone’s private network. In the baseline, the client sends data to help with this identification, sending the SIM’s ICC-ID and IMSI, which also confirm the client’s identity.
Figure 10 depicts the bytes received by the client in the “Get Token” flow. As in Figure 9, it is possible to see a decrease in bytes. Again, the decrease of bytes received in the physical layers is explained by the reduced number of packets received. In the network layer, it is explained by the use of UDP instead of TCP and, in the application layer, it is explained by the use of CoAP instead of HTTP. The reduction in data sent is explained by the usage of a CWT instead of a JWT to codify the access token.

2) *“Client Publish” Flow:* Now the depiction of the “Client Publish” flow. Table III shows the bytes received and sent by the client, and further detailed per OSI layer in Figures 11 and 12
In this case, the decrease in traffic network is not so meaningful with the changes implemented in the first intermediate solution. Most changes were targeted at the “Get Token” flow, however, there is an 18.67% decrease in bytes sent by the client due to the changes to the token format, meaning that the token is

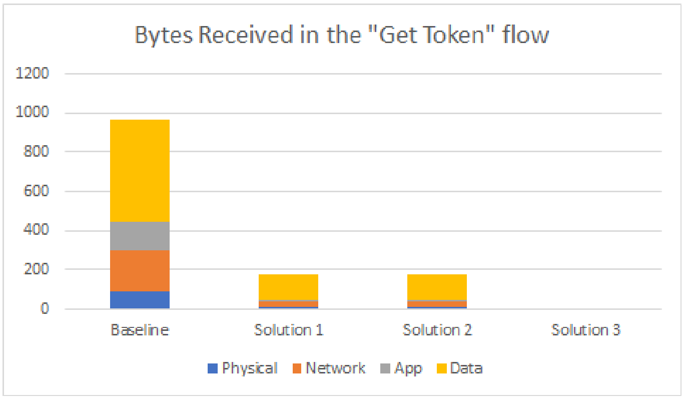

Fig. 10. Bytes Received in the “Get Token” flow

TABLE III
BYTES RECEIVED AND SENT BY THE CLIENT IN THE “CLIENT PUBLISH” FLOW FOR EVERY SOLUTION

	Baseline	Solution 1	Solution 2	Solution 3
Received	6204.6 ± 37.757	6169.4 ± 46.419	52 ± 0	52 ± 0
Sent	1859 ± 25.456	1512 ± 25.456	228 ± 0	55 ± 0

smaller and so the client sends fewer data. There is a meaningless reduction of 0.57% that is statistically irrelevant. The significant changes come with the second intermediate solution that eliminates the costly use of MQTTS with the introduction of the broker. With a whopping 99.16% reduction in received bytes received and an 87.74% reduction in sent bytes, when compared to the baseline, showing that the intermediate solution 2 presents clear benefits to the client. The proposed solution eliminates the need to send the token to the broker which helps to enhance this gain from 87.74%, when compared to the baseline, to 97.04%.

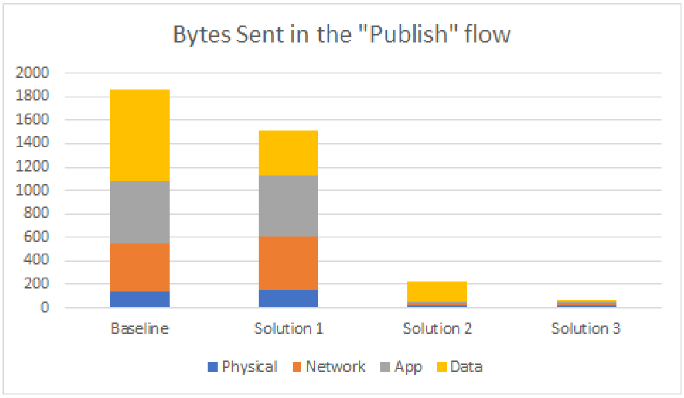

Fig. 11. Bytes Sent in the “Client Publish” flow

In Figure 11 there is more nuance as there is a difference in performance, not only between the baseline and the intermediate solutions but also between the intermediate solutions themselves. The major difference between the baseline and the first intermediate solution is in the data which comes from the difference in technology used to codify the token. The use of CWT instead of JWT is responsible for this difference. The big changes are seen when comparing the first to the second intermediate solution, given the introduction of the broker. The

introduction of the broker allows for the use of CoAP instead of using MQTTS. The use of MQTTS, which requires various handshakes and a heavy overhead, is replaced by a single CoAP request and this is apparent in the performance of the two prototypes. The proposed solution optimizes this flow since there is no need to send the token in the request.
In Figure 12 the difference in performance is explained by the fact that both the baseline and first intermediate solution use MQTTS and have to perform costly TCP and TLS handshakes where as the intermediate solution 2 and the proposed solution use CoAP. The big amount of data, when compared to other layers, is due to the need to receive the TLS certificate of the third-party.

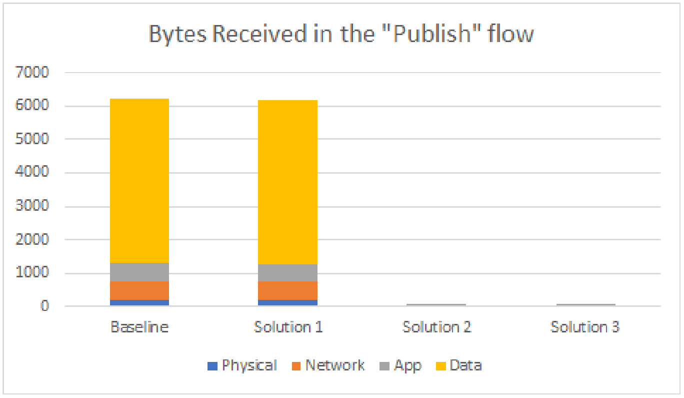

Fig. 12. Bytes Received in the “Client Publish” flow

D. Evaluation

The results presented show that the proposed solution brings a massive increase in performance, with the intermediate solution 2 and the final proposed solution showing that the introduction of the broker comes with tremendous benefit for the device, as it allows the device to use CoAP for every communication effectively bypassing the third-parties restrictions. Another important analysis is to check if these prototypes are ready to use NB-IoT. An NB-IoT device is expected to transmit up to 200 bytes per day. In table IV the total network traffic of the client is depicted.

TABLE IV
TOTAL TRAFFIC NETWORK IN THE CLIENT FOR EVERY SOLUTION

	Baseline	Solution 1	Solution 2	Solution 3
Total Traffic Network	9546.8 ± 36.199	7912 ± 52.679	517 ± 0	107 ± 0

Assuming that a client needs to send a message to the third-party at least one time per day the only solution that is ready to use NB-IoT is the final proposed solution. The message sent in the tests was a 3-byte message which means that the total overhead for communications is 104 bytes, allowing the client to send one message per day to the third-party up to 96 bytes.

VI. Related Work

Gerber [6] implements the ACE framework, analysing it’s efficiency when compared to OAuth 2.0, but runing the protocol over HTTP instead of CoAP.

Truphone’s Existing Solution implements an authentication mechanism for IoT devices based on the authentication made by the SIM in cellular networks.
Lagutin et al. [15] use the ACE framework to allow constrained IoT devices to use verifiable credentials and decentralized identifiers to manage authentication and authorization.

VII. Conclusion

In this work, a new innovative way to authenticate constrained devices operating in constrained environments to a third-party is presented. It relies on cellular authentication provided by Truphone and on the introduction of a broker to which the client delegates all authentication mechanisms. The proposed solution fulfils provides the client with authentication based on his cellular connection, it reduces traffic network in the client by more than 50% and it allows for a seamless authentication mechanism, as the client only needs to send a message to the broker and the broker will deal with the authentication to the third-party.

References

[1] S. Jaiganesh, K. Gunaseelan, and V. Ellappan, “Iot agriculture to improve food and farming technology,” in *2017 Conference on Emerging Devices and Smart Systems (ICEDSS)*. IEEE, 2017, pp. 260–266.
[2] L. S. Chandana and A. R. Sekhar, “Weather monitoring using wireless sensor networks based on iot,” *Int. J. Sci. Res. Sci. Technol*, vol. 4, pp. 525–531, 2018.
[3] M. Tu, “An exploratory study of internet of things (iot) adoption intention in logistics and supply chain management,” *The International Journal of Logistics Management*, 2018.
[4] D. Hardt, “The oauth 2.0 authorization framework,” Internet Requests for Comments, RFC Editor, RFC 6749, October 2012, http://www.rfc-editor.org/rfc/rfc6749.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt
[5] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig, “Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth),” Internet Engineering Task Force, Internet-Draft draft-ietf-ace-oauth-authz-42, Jun. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ace-oauth-authz-42
[6] U. Gerber, “Authentication and authorization for constrained environments,” Master’s thesis, University of Zurich, 2018.
[7] G. Selander, J. Mattsson, and F. Palombini, “Object security for constrained restful environments (oscore),” Internet Requests for Comments, RFC Editor, RFC 8631, July 2019, https://www.rfc-editor.org/rfc/rfc8631.txt. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8631.txt
[8] E. B. Barker and Q. H. Dang, “Recommendation for key management part 3: Application-specific key management guidance,” Tech. Rep., 2015. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-57Pt3r1
[9] S. Nakov, *Practical Cryptography for Developers*. Gitbook, 2018, https://cryptobook.nakov.com/.
[10] E. Barker, “Digital signature standard (dss),” October 2019.
[11] J. de Carvalho Silva, J. J. Rodrigues, A. M. Alberti, P. Solic, and A. L. Aquino, “Lorawan—a low power wan protocol for internet of things: A review and opportunities,” in *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*. IEEE, 2017, pp. 1–6.
[12] D. Darwich, “Comparison between java, go, and rust,” https://medium.com/@dexterdarwich/comparison-between-java-go-and-rust-fdb21bd5fb7c, accessed: 2021-03-26.
[13] https://github.com/Covertness, “coap,” https://crates.io/crates/coap.
[14] https://github.com/pyfisch, “serde_cbor,” https://crates.io/crates/serde_cbor.
[15] D. Lagutin, Y. Kortesniemi, N. Fotiou, and V. A. Siris, “Enabling decentralised identifiers and verifiable credentials for constrained iot devices using oauth-based delegation,” in *Proceedings of the Workshop on Decentralized IoT Systems and Security (DISS 2019), in Conjunction with the NDSS Symposium, San Diego, CA, USA*, vol. 24, 2019.