Securing Complex IoT Platforms with Token Based Access Control and Authenticated Key Establishment

Timothy Claeys, Franck Rousseau, Bernard Tourancheau

▶ **To cite this version:**

Timothy Claeys, Franck Rousseau, Bernard Tourancheau. Securing Complex IoT Platforms with Token Based Access Control and Authenticated Key Establishment. International Workshop on Secure Internet of Things (SIOT), Sep 2017, Oslo, Norway. hal-01596135

HAL Id: hal-01596135
https://hal.science/hal-01596135

Submitted on 9 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Securing Complex IoT Platforms with Token Based Access Control and Authenticated Key Establishment

Timothy Claeys, Franck Rousseau, Bernard Tourancheau
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France
Email: firstname.lastname@imag.fr

***Abstract*—In this paper we propose a new authorization and authentication framework for the IoT that combines the security model of OAuth 1.0a with the lightweight building blocks of ACE. By designing self-securing tokens the security of the framework no longer depends on the security of the network stack. We use basic PKI functionalities to bootstrap a chain-of-trust between the devices which simplifies future token exchanges. Finally, we propose an alternate key establishment scheme for use cases where devices cannot directly communicate.**

We test our proposal by implementing the critical aspects on a STM32L4 microcontroller. The results indicate that our framework guarantees a strong level of security for IoT devices with basic asymmetric cryptography capabilities.

***Keywords*-authorization; access tokens; key establishment; authentication**

I. Introduction

In recent years we have seen a profusion of Internet of Things (IoT) platforms [1]. These IoT platforms consist of heterogeneous, often constrained, devices with complex network stacks. The design of security services that provide fine-grained access control, authentication and confidentiality are a challenge for most devices in the IoT. To protect data from unauthorized access, standard solutions use Public Key Infrastructure (PKI), Transport Layer Security (TLS), user passwords, and token architectures. These solutions are often ill-suited for IoT architectures as they have a large computational overhead and require ubiquitous connectivity of the smart devices.

The IEFT ACE working group has proposed a generic framework suitable for authentication and authorization in constrained environments. ACE is based on OAuth 2.0. It enables a third party, defined as the *client*, to access protected resources from a *resource server*, e.g. a smart device. Instead of requiring the owner of the device, the *resource owner*, to disclose his or her credentials, access is regulated by tokens provided by a trusted third party, the *authorization server*. The generic architecture of ACE covers a wide range of possible use cases with different requirements from a security point of view [2]. ACE defines profiles that describe the transport mechanisms and security protocols for different deployments. For example, clients that directly access the resource server use ACE either in combination with the Datagram Transport Layer Security (DTLS) [3] or with Object Security of COAP (OSCOAP) [4]. Clients in publish-subscribe architectures can use the ACE pub-sub profile.

The main drawback of both ACE and OAuth 2.0 is that both depend on the security of the underlying communication protocols, e.g. (D)TLS, and a trusted third party, the authorization server. The authentication of the different entities, resource owner, resource server and authorization server, happens with pre-shared secrets. ACE does not provide the option for dynamic entity authentication with PKI. For large scale deployments, the pre-distribution of entity authentication secrets might be infeasible.

In this paper, we present a new secure authorization and authentication framework that addresses the main security issues of ACE. We follow the security model of OAuth 1.0a [5]. Our framework uses self-securing tokens and is therefore independent from the security properties of the underlying network stack. This makes it suitable for complex, multi-hop environments. In these contexts the underlying network stack changes and does not always guarantee secure communication. We introduce basic PKI functionalities in our achitecture. This removes the need to blindly trust the authorization server. Moreover, we capitalize on an initial trust anchor between the client and the resource server by setting up a chain-of-trust. The chain of trust allows the client to reuse tokens while protecting them with simple symmetric cryptographic functions. Yet, we leverage a vast amount of work behind the ACE architecture. More precisely, we keep the ACE authorization flow and adopt the concept of proof-of-possession tokens that binds the identity of a client with an access token. Our tokens are represented with the Concise Binary Object Representation (CBOR) [6] format and protected using Concise Object Signing and Encryption (COSE) objects [7].

We then propose an alternative for Ephemeral Diffie Hellman over Cose (EDHOC) as Authenticated Key Establishment (AKE) scheme. A shared secret between the client and resource server is calculated with the Elliptic Curve Integrated Encryption Scheme (ECIES) [8]. ECIES can be used in all ACE's profiles. However, it is optimized for use cases where the client and resource server cannot directly communicate. For example, when two nodes in different duty-cycled sensor networks want to communicate, their messages must be buffered in a safe way in a network proxy.

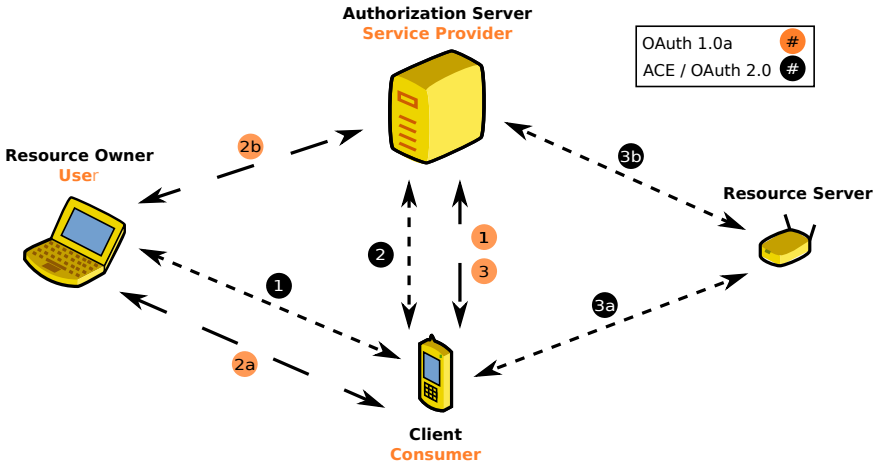

Figure 1. The OAuth 1.0a and ACE authorization flows.

We have tested the critical aspects from our architecture on a STM32L4 [9]. The results for cryptographic primitives show that after an initial expensive token verification phase, further access control with the same token can be done with negligible overhead. Furthermore, we measured the memory overhead for the cryptographic primitives, which amounts in total to 45kB. We also measured the token sizes and compared them with the maximum packet length of Bluetooth Low Energy, IEEE802.15.4 and NFC.

The rest of the paper is organized as follows: In Section II we provide a background on ACE, OAuth 1.0a and their security models. Section III defines the threat model for our framework. In Section IV and V we propose our framework and test its properties. A security analysis is given in Section VI and Section VII concludes the paper.

II. ACE vs. OAuth 1.0a

A. OAuth 1.0a

The OAuth protocol was designed for use on the web and uses HTTP for commmunication. It enables websites or applications (consumers) to access protected resources from a web service (service provider). The consumers don't require users to disclose their credentials [5]. The OAuth protocol has 3 phases, depicted in Figure 1. In phase 1, the consumer asks an *unauthorized request token* from the service provider (1). In phase 2, the consumer requests the authorization of the user to access the protected resources (2a). The consumers redirects the user to the service provider. The user authenticates him or herself, e.g. using normal user and password credentials, and then grants or denies the authorization request of the consumer (2b). If the user accepts, the consumer is informed that its unauthorized access token from phase 1 has been authorized. In phase 3, the consumer returns to the service provider and exchanges its authorized request token for an access token. The access token can be used to access the protected resources (3).

B. OAuth 1.0a Security

OAuth 1.0a requires consumers to generate new signatures on every request. These can be calculated through asymmetric cryptography or Hashed Message Authentication Codes (HMAC) and provide integrity and authenticity of the requests. Generating and validating signatures creates a considerable overhead as parameters have to be parsed, sorted, and hashed in a certain way for every request. The secrets needed for signing are provided during an enrollment phase and are then never transmitted across the wire. OAuth 1.0 does not assume an underlying secure channel. Because of these reasons OAuth 1.0a is considered more secure than OAuth 2.0 but more difficult to implement [1].

C. ACE Framework

The ACE architecture introduces a flexible authorization and authentication for the IoT. The current specification is based on OAuth 2.0. It uses CoAP with CBOR encoded messages instead of HTTP. ACE uses the naming convention of OAuth 2.0, but the entities act primarily the same as in OAuth 1.0a. The resource owner replaces the user. The protected resources are now stored on a newly introduced entity, the resource server, instead of the service provider. The client has the role of consumer. The ACE flow starts with the client obtaining the resource owner's consent, see Figure 1. The consent can be delivered dynamically as in the OAuth schemes or it can be pre-configured by the resource owner at the authorization server (1). In step 2, the client makes an access token request to the authorization server. If the authorization server successfully processes the request from the client, it returns an access token (2). In the final step, the client presents the token to the resource server. If the resource server can validate the token, access is granted to the protected resource (3a). The resource server can try to validate the tokens locally or it will request a *token introspection* at the authorization server. The authorization

server will then verify the token for the resource server (3b). Tokens that can be locally verified are called *self-contained tokens*.

D. ACE Security

Similarly to OAuth 2.0 that uses TLS to protect HTTP traffic, ACE uses secure communication protocols such as DTLS and Object Security to protect CoAP messages. The CoAP traffic carries both the tokens and the protected resources. The entities in ACE always use mutual authentication. This is obtained through pre-shared raw public keys or symmetric keys. Authentication through PKI is not supported in ACE. ACE defines an additional defense against token theft. It introduces Proof-of-Possession (PoP) tokens. To generate PoP-tokens the authorization server binds cryptographic keys to the traditional access tokens. These so called PoP-keys can be symmetric or asymmetric key pairs. Symmetric PoP-keys are randomly generated by the authorization server. The authorization server shares one copy with the client. A second copy of the PoP-key is either stored by the authorization server to be used on token introspection or securely shared with the resource server so that it can locally verify the PoP-tokens. If the client uses asymmetric PoP-keys, the authorization server binds the public key of the client to the PoP-token. The client uses the symmetric or asymmetric PoP-keys to demonstrate the possession of a secret to the resource server when accessing the protected resource through the presentation of its PoP-token. It proves that the client is the valid owner of the token.

1) CoAP over DTLS: When using DTLS as secure transport layer the client will set up a secure channel with the resource server [3]. The client can either use asymmetric cryptography with raw public keys or a symmetric pre-shared key mode. In the first scenario a client publishes its raw public key to the authorization server. The authorization server generates a token and binds it to the client's public key. Before starting the DTLS handshake the client must send its token to the resource server. The resource server must obtain the public key of the client that was bound to the token. The resource server can then authenticate the client. While in pre-shared key mode, the client and resource server can use a pre-shared symmetric pop key, provided by the authorization server, to mutually authenticate and secure the DTLS session.

2) OSCOAP: In contrast to DTLS, OSCOAP does not protect the communication channel but uses COSE objects to protect the payload of a CoAP message end-to-end, across intermediary nodes [4]. OSCOAP uses a common security context between the client and the resource server. This context specifies a a pre-established master secret and an Authenticated Encryption with Additional Data (AEAD) algorithm that is used to encrypt and protect the payload. Similarly to DTLS, a symmetric PoP-key, provided by the authorization server, can be directly used as a master secret to derive OSCOAP's security contexts and mutually authenticate the client and the resource server. Alternatively, a symmetric or asymmetric PoP-key can be used to authenticate the messages during EDHOC. The secret derived from EDHOC [10] is then used as master secret for the security context.

III. Threat model

In order to design a token-based authorization and authentication scheme we must analyze the different threats. The threat model lists token-specific threats and threats against entities in the architecture. The token-related threats are:

- Integrity of the tokens: we want to prevent unauthorized entities from forging valid tokens or changing the content of the token after its creation.
- Token theft: a token theft detection mechanism must be in place to prevent unauthorized entities from using stolen tokens.

The threats against entities in the architecture are presented below:

- The resource servers in IoT architectures are often constrained devices. They can be deployed in remote areas and can lack basic security features, such as memory protection units (MPU). These devices are prone to a wide range of network and physical attacks. Because of these reasons we must minimize the storage of secrets in the resource servers and verify their identity during the token exchanges.
- The clients can be very heterogeneous. They can be laptops, smartphones, cloud services or other smart devices. The tokens stored on these devices must be protected from theft. The clients must be authenticated during the token exchanges and we must prevent them from forging valid tokens.
- The authorization server creates tokens for the clients after they successfully authenticate. In our new architecture we want to prevent a compromised authorization server from accessing all the protected resources.

IV. Proposed Authorization Architecture

We design a new token-based access control scheme with authenticated key establishment for IoT platforms. The access tokens consist of a set of encoded access rights, known as token claims, wrapped in a COSE object. Our architecture is standalone and does not depend on the security of the underlying transport method. It handles resource servers with intermittent Internet connectivity by using self-contained tokens. Clients can be provided with long-lived tokens if they have limited connectivity with the authorization server. Long-lived tokens allow multiple authentications with the same token. The main advantage is that our framework can easily be used in complex, multi-hop environments where the underlying network stack changes and therefore

does not always guarantee secure communication. We avoid the use of trusted third parties. The framework uses the same name conventions as the ACE specification. We make several assumptions about the entities in our framework: the constrained devices are capable of lightweight asymmetric cryptography and every device can either contact a Certificate Authority or has a root certificate onboarded. This allows the participating devices to sign messages and verify the identities behind the signatures.

A. Access Token Generation

Before a client can recover a valid access token for a protected resource it must obtain permission from the resource owner. With a valid permission it can request an access token from the authorization server. The authorization server sets the token claims according to the permission and wraps the token claims in a *COSE-sign* object, shown in Figure 2. The authorization server signs the protected header and the payload, and transfers it to the client. The client verifies the authorization server's signature and signs the COSE object with its own private key.

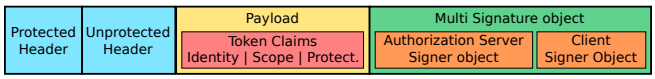

Figure 2. The self-securing token based on CWT specification. The token claims are wrapped in a COSE-sign object, allowing for multiple signatures

The token claims use the same structure as the CBOR web token (CWT) [11] specification. It consists of three parts: the token identity information fields, token scope fields and token protection fields.

1) Token Identity Information: The token identity information fields contain a *subject field* and an *audience field*. The first identifies a unique resource server that is the target of this token. The latter stores the client's public key or a key identifier of the client's public key for which the token is created. The full public key can be presented by a COSE key object. The *issuer field* defined in CWT is omitted in our token model because a key identifier for the public key of the authorization server is already included in the signature part of the enveloping COSE object.

2) Token Scope: The scope of the token is defined in the *resource scope field*. It describes the different actions a client is allowed to take once the token gets validated by the resource server. These actions are application dependent. The *expiration*, *not-before* and *issued-at fields* store timing information on when the token can be used and when it was created.

3) Token Protection: The token protection field holds two replay counters. The *long-term replay window* is responsible for replay protection in between different tokens. The window contains an integer that is incremented by the authorization server each time a client requests a fresh token from the authorization server. The authorization server maintains per client a long-term replay window value that is independent from other clients. The *short-term replay window* is a replay protection window, the scope of which is limited to the lifetime of the token. A resource server stores, per client, both replay window values. The value of the short-term replay window must change on every token use to obtain the PoP-principle, see Section IV-B.

When the resource server uses the token timing information, such as the expiration, not-before and issued-at fields, to limit the token usage, the short-term replay window is a simple counter that increments for every token use. If the resource server does not possess precise time-keeping hardware, the authorization server leaves the expiration field in the token scope blank and sets the short-term replay window to a specific value during the token creation. The resource server then decrements the short-term window for every token use until the window value reaches 0. At 0 the token is no longer valid and the client will have to request a new token at the authorization server.

B. Authenticating to the Resource Server

In deployments without an intermediary network proxy, the client sends its token directly to the resource server, see (1) in Figure 3. A token is always processed in two phases. In the first phase the token is validated. When a client authenticates with a fresh token, the resource server verifies the following information:

1) It verifies that the token hasn't expired. It either checks the token scope field or the short-term replay window.
2) It verifies the signatures of the authorization server and the client.
3) If the client has no previous records, the resource server uses the audience field to create a client ID. It then stores the replay values for this ID.

In phase 2, the resource server creates a *COSE-encrypt* object. It copies the token claims from the received *COSE-sign* object and updates the short-term replay protection window. It encrypts the payload according to the COSE specification. The key for the encryption algorithm, K_{sh}, is derived from a shared secret established during the AKE. The AKE takes places after the initial token verification, see Section IV-C. The resource server then protects the integrity of the payload, containing the token claims, and the protected header fields of the COSE-encrypt object by calculating a HMAC according to the COSE specification. The key necessary for the HMAC is generated locally on the resource server and only known to the resource server, K_{rs}. It prevents the client from tampering with the token, i.e. changing the token claims. The resource server then sends the encrypted updated token back to the client, see (2) in Figure 3.

These initial two steps are expensive due to the asymmetric cryptography needed to verify a fresh token and

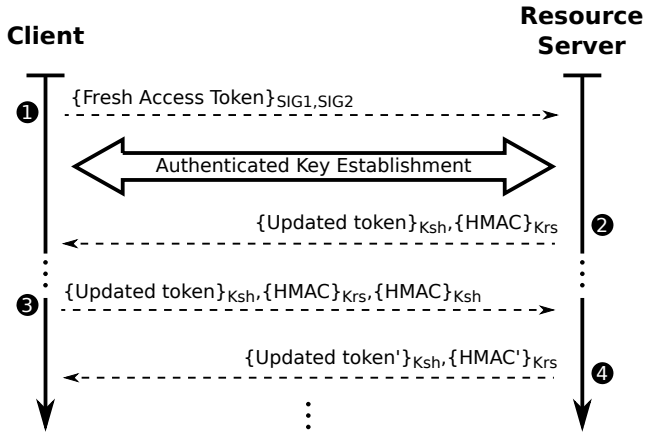

Figure 3. The token exchanges between the client and the resource server: SIG1 and SIG2 are the authorization server and client signatures. K_{sh} and K_{rs} are the shared key and the resource server HMAC's key, respectively

the setup of the shared secret with the AKE protocol. The verification of a fresh token bootstraps a *chain-of-trust* that allows the client and resource server to use solely symmetric cryptography in the following token verifications.

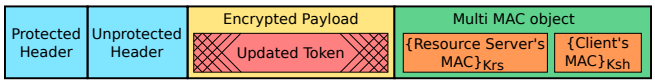

Figure 4. The updated token claims, wrapped in a COSE-encrypt object. We use the recipients fields of the COSE-encrypt object to put an integrity check of the resource server and an integrity check of the client in the COSE object.

The client then receives the updated token. It can verify the origin because the token was encrypted with the shared key, K_{sh}. The shared key thus acts as a PoP-key. The next time the client wants to authenticate to the resource server, it uses the received token. Before it sends the token to the resource server for a new authentication, it reencrypts the token and calculates its own HMAC over the payload, with K_{sh}, see (3) in Figure 3. A verifiable HMAC proofs knowledge of the PoP-key to the resource server. The token is depicted in Figure 4. The resource server again uses 2 phases to process the token. It now verifies the following information:

1) It verifies the token's expiration time.
2) It verifies its own HMAC over the token, with K_{rs}, to detect tampering.
3) It verifies the client's HMAC over the token, with K_{sh}, to validate its identity.
4) It checks the validity of the long-term token replay number and the short-term replay number in the token by comparing it to its local versions.

If all checks are valid, the token originates from the rightful client and it has not been tampered with. Because with each token use, the resource server updates the short-term replay window, the integrity tags change every time. A man-in-the-middle (MITM) or replay attack is therefore not possible. Before sending back the updated token to the client, the resource server removes the client's HMAC. The client can then again prove it's knowledge of the shared secret by calculating the HMAC over the updated token, see (4) in Figure 3.

C. Authenticated Key Establishment

Similarly to ACE, we use EDHOC to establish a shared secret between the client and the resource server. EDHOC is authenticated with the private keys from both the client and the resource server bound to their respective certificates. This provides a mutual authentication during the shared secret establishment. The shared secret is then used to derive two symmetric keys. The first key, K_{sh}, is used as a PoP-key to proof valid ownership of the token. The second key is used in combination with an AEAD scheme to protect the resources.

In scenarios where a direct connection between the resource server and client is not possible, we propose the use of Elliptic Curve Integrated Encryption Scheme (ECIES) as the key establishment scheme. Where EDHOC uses 2 messages to set up a symmetric key, ECIES can function as an offline scheme. The resource server derives a symmetric key based on the public key of the client and a locally generated ephemeral key pair. With this key it can encrypt data and securely store it in the proxy server before the client has derived its key. An example where a direct connection might not be possible is when both the client and resource server are sensor nodes that belong to two distinct wireless networks. The nodes sleep most of the time and their duty cycles are not synchronized. Authentication and key establishment between both entities can be solved in the following way, see Figure 5. The client posts a fresh access token to a proxy (1). The proxy forwards the token to the resource server when it becomes available (2). The resource server validates the token locally. It then starts the ECIES scheme. The resource server generates an ephemeral key pair $U = u \cdot G$, where u is the private key, G the generator of the elliptic curve and U the public value. It selects the Diffie-Hellman key agreement function and provides its private key, u, and the client's public key extracted from the token, V, as input. The resulting secret is $u \cdot V$. It can combine the secret with optionally additional data and feed it to a Key Derivation Function (KDF). The KDF is run until a key for encryption and a key for the MAC can be derived. Both keys are used in combination with an AEAD scheme to encrypt the resources posted to the proxy and protect their integrity (3). The resource server then pushes the encrypted resource, updated token and its ephemeral public key to the network proxy. The resource server signs the ephemeral public key, $\{U\}_{SIG}$, with its static private key bound to its certificate before it is uploaded to the proxy. When the client connects

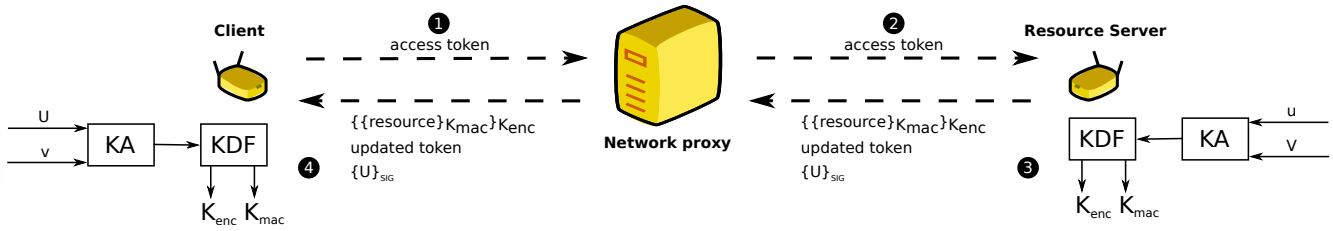

Figure 5. Authenticated key establishment over an indirect connection using ECIES.

to the network proxy, it retrieves the encrypted resources, its updated token and the signed ephemeral public key of the resource server. The signature on the ephemeral public key allows the client to validate the resource server's identity. It subsequently runs its own instance of the ECIES scheme to derive the same symmetric keys. The client can now decrypt and verify the integrity of the retrieved resources.

The next time a client wants to access the protected resources it uploads its new updated token. After verification of the token the resource server can decide to use the old derived keys or rerun the ECIES scheme with a new ephemeral key pair. The latter provides Perfect Forward Secrecy (PFS), similar to EDHOC.

V. Implementation

In this section we measure the critical aspects of our framework.

A. Computational Overhead

We tested the cryptographic primitives, used in our architecture, on a STM32L4 [9]. The STM32L4 uses a Cortex-M4 at 80 MHz as processor. The results are depicted in Table I. We used mbed TLS [12] [13] as cryptographic library. All the elliptic curve operations used the optimized NIST curves and are supported by COSE. The resource server starts the token processing with the validation of 2 signatures.

Table I
Cryptographic overhead (ms)

Primitive	Encrypt/Sign	Decrypt/Verify
AES (128 bit)	0.079	0.155
HMAC (SHA256)	0.386	-
ECDSA (P-256)	248	839
Key gen. (P-256)	590	-
ECDH (P-256)	581	-

Afterwards it needs to set up the shared secret using either EDHOC or ECIES. We can see that this has a significant cryptographic overhead. The first verification and setup is expensive but bootstraps a *chain-of-trust* between the client and the resource server. Once the initial verification is completed and EDHOC has derived an authenticated key, the other token exchanges use solely symmetric cryptography.

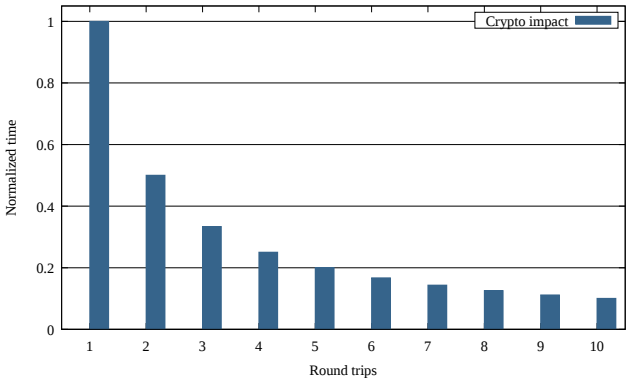

Figure 6. The impact of the cryptographic overhead from the resource server's point of view. The initial overhead becomes less important by every updated token exchange. The exchange of an updated token uses only symmetric cryptography.

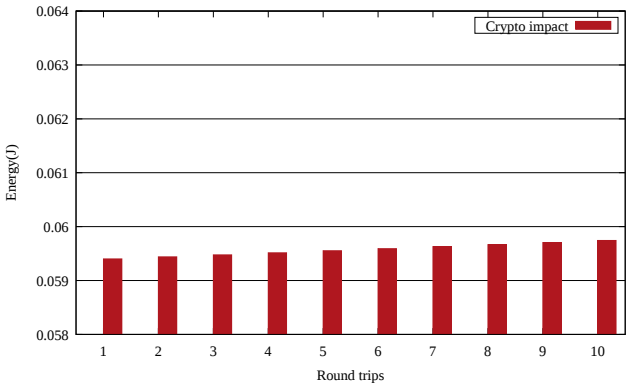

Figure 7. The impact of the cryptographic overhead from the resource server's point of view. The initial token verification uses a significant amount of energy but the additional energy consumed by the symmetric cryptography is negligible.

We didn't test the ECIES scheme because at its core it uses the same primitives as EDHOC. The normalized impact of the cryptographic operations per token exchange is depicted in Figure 6 and the additional energy usage per token exchange is shown in Figure 7.

B. Memory Overhead

The memory overhead of our scheme can be split up in two parts: the code size and the storage of client information.

1) Code Size: We measured the code size of the cryptographic primitives in the mbed TLS on the STM32L4. The results are depicted in Table II. While the code size of the symmetric primitives is relatively small, both ECC primitives have a distinct overhead. A part of ECDSA's code size is function calls to the mbed TLS hash library. The total code size for the cryptographic primitives amounts to 45 kB. The resource server and client also use the CBOR and COSE encoding libraries for the encoding and formatting of the tokens.

Table II
MEMORY OVERHEAD OF THE CRYPTOGRAPHIC PRIMITIVES (BYTES)

Primitive	Code size
AES (128 bit)	4500
HMAC (SHA256)	5304
ECDSA (P-256)	26676
ECDH (P-256)	17340

2) Client Data and Key Storage: The resource server stores for each client the long-term and short-term replay windows. These are linked to the client's ID which can be a hash of the client's public key. The resource server also stores the shared symmetric key for encryption of the updated token. The total memory overhead per client is 448 bits if we take 32 bits for both replay windows. The system-wide stored information is the certificate of the authorization server, the resource server's root certificate, and the locally derived key, K_{rs}, to calculate the HMAC over the updated tokens. This key can be used to protect the integrity of the all the tokens used by the different clients.

C. Bandwidth Limitations

Constrained devices use communication protocols with limited packet sizes, such as Bluetooth Low Energy (BLE), IEEE802.15.4 and Near Field Communication (NFC). The maximum packet sizes of these protocols are 245 bytes, 120 bytes and 254 bytes, respectively [14] [15]. The size of the exchanged tokens is therefore an important factor. We implemented our own Python COSE library and tested the COSE object sizes for our different token types: the fresh token and the updated token. We used an example token for test purposes:

- Subject field: hash of the resource server's public key (256 bit)
- Audience field: hash of the client's public key (256 bit)
- Scope: a 32 bit field where every bit represents a protected resource on the resource server.
- Long-term replay window: 32 bit
- Short-term replay window: 32 bit. No timing information is set in the token scope, so this field contains the total amount of allowed token uses.

The total raw token size is 74 bytes. The token encoded as a CBOR array amounts to 80 bytes. The COSE-sign object has the token as payload, two signatures and the signature algorithms and key identifiers in the protected header. The total size of the COSE-sign object of the fresh token is 245 bytes. The largest part of the COSE-sign object are the signatures calculated with ECDSA and P-256 as curve. COSE supports P-256, P-384 and P-512. When the client reauthenticates with the updated token the COSE object has been transformed to a COSE-encrypt object. It contains the CBOR-encoded token as encrypted payload, 2 HMACs and information on the cryptographic primitives and key identifier in the protected headers. The total size of the updated token is 181 bytes. We used SHA-256 as internal hash function. Every HMAC has thus a total size of 32 bytes. COSE supports HMAC lengths of 64, 256, 384 and 512 bits.

We notice that the COSE-mac object fits in a BLE and NFC frame. The COSE-sign does not fit in the frames, because the underlying network protocols also add their headers before the packet is sent. When using tokens in wireless sensor networks, which mainly use IEEE802.15.4, the tokens will always have to be split up over several packets.

VI. SECURITY CONSIDERATIONS

This section discusses security aspects of our proposed architecture.

A. Protection against Eavesdropping Attacks

The fresh token, signed by both the authorization server and the client, is not encrypted during transmission. Attackers can thus read the content of the token. This does not pose a security issue as the token does not contain any secrets. In case there are privacy concerns, the fresh token can be sent over a secured channel. After the authenticated key establishment the updated tokens and the exchanged resources are encrypted with the shared key, established during the AKE.

B. Protection against Replay Attack

The tokens contain a long-term and short-term replay window. The long-term window provides protection against replay when an attacker tries to reuse an older token. This replay window only increases when a specific client obtains a new token from the authorization server. The value is set by the authorization server and checked and stored by the resource server. The short-term replay window provides protection when a client uses the same token more than once. The value starts at 0 for a fresh token and is incremented by the resource server on every usage. Notice that the resource server must store and update the replay window values, otherwise a replay attack is possible.

C. Protection against Man-in-the-Middle Attack

Man-in-the-middle attacks are thwarted by the use of signatures or HMACs. A fresh token carries two signatures. One from the authorization server and one from the client. The resource server uses its onboarded certificates to validate the identities. While exchanging the updated tokens we employ the proof-of-possession concept from ACE. The shared key K_{sh}, derived from the AKE, is used by the client to calculate a HMAC and encrypt the updated token. The resource server only uses the shared key for encryption. It calculates its own HMAC with its local key, K_{rs}. Correct usage of K_{sh} implies knowledge of the shared secret.

D. Protection from a Rogue Client

A malicious client cannot forge tokens. A fresh token must have a valid signature from the authorization server. The integrity of the updated tokens is protected by the HMAC calculated by the resource server with K_{rs}. This key is only known to the resource server.

E. Protection against Compromised Authorization Servers

Fresh tokens are only considered valid when they have signatures of both the client and authorization server. Because the signatures are validated through the certificate system an authorization server cannot easily spoof the signature of a client.

F. Protection against Compromised Resource servers

If a resource server were compromised, all the stored cryptographic material could be revealed. We advise that resource servers that are capable use the PFS modes of EDHOC and ECIES. This minimizes the amount of compromised data when the resource server is captured.

VII. Conclusion

This paper proposes a new authentication and authorization framework. The main idea is to combine the security model of OAuth 1.0a with the lightweight IoT architecture of ACE. We designed a framework that uses basic PKI functionalities for dynamic entity validation and self-securing tokens. The tokens can be long-lived and are self-contained. This allows for deployments where both the client and resource server have only limited connectivity with the authorization server. The security of the tokens does not depend on the security of the underlying transport protocols and avoids the use of trusted third parties. We achieve this by using signatures on the fresh tokens obtained from the authorization server and HMACs on the updated tokens. We also propose a new authenticated key establishment scheme. ECIES allows for offline key derivation which is useful in use cases where the client and resource server can't directly communicate.

The experimental results show that the initial verification of the fresh tokens is expensive but it establishes a chain-of-trust. This can be leveraged by applications to strongly reduce the time and energy needed to verify the same token in the future.

VIII. Acknowledgments

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir, the FUI IoTize project funded by Région Auvergne-Rhône-Alpes, and the DataTweet project under contract ANR-13-INFR-0008-01.

References

[1] A. Niruntasukrat, C. Issariyapat, P. Pongpaibool, K. Meesublak, P. Aiumsupucgul, and A. Panya, "Authorization mechanism for MQTT-based Internet of Things," in *Communications Workshops (ICC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 290–295.

[2] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE)," IETF, Internet-Draft draft-ietf-ace-oauth-authz-06, Mar. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ace-oauth-authz-06

[3] S. Gerdes, O. Bergmann, C. Bormann, G. Selander, and L. Seitz, "Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE)," IETF, Internet-Draft draft-ietf-ace-dtls-authorize-00, Jun. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ace-dtls-authorize-00

[4] L. Seitz, M. Gunnarsson, and F. Palombini, "OSCOAP profile of ACE," IETF, Internet-Draft draft-seitz-ace-oscoap-profile-02, May 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-seitz-ace-oscoap-profile-02

[5] (2016, June) OAuth Core 1.0 Revision A. OAuth Core Workgroup. [Online]. Available: https://oauth.net/core/1.0a/

[6] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 7049, Oct. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc7049.txt

[7] J. Schaad, "CBOR Object Signing and Encryption (COSE)," IETF, Internet-Draft draft-ietf-cose-msg-24, Nov. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-cose-msg-24

[8] V. G. Martínez, L. H. Encinas, and C. S. Ávila, "A Survey of the Elliptic Curve Integrated Encryption Scheme," *ratio*, vol. 80, no. 1024, pp. 160–223, 2010.

[9] *STM32L432KC STM32L4KB*, STMicroelectronics, June 2017, rev. 3.

[10] G. Selander, J. Mattsson, and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)," IETF, Internet-Draft draft-selander-ace-cose-ecdhe-06, Apr. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-selander-ace-cose-ecdhe-06

[11] M. Jones, H. Tschofenig, E. Wahlstroem, and S. Erdtman, "CBOR Web Token (CWT)," IETF, Internet-Draft draft-ietf-ace-cbor-web-token-05, Jun. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ace-cbor-web-token-05

[12] (2016) ARM mbed: mbed TLS. ARM. [Online]. Available: https://tls.mbed.org

[13] (2015) Nist lightweight cryptography workshop. ARM. [Online]. Available: http://csrc.nist.gov/groups/ST/lwc-workshop2015/presentations/session7-vincent.pdf

[14] É. Morin, M. Maman, R. Guizzetti, and A. Duda, "Comparison of the Device Lifetime in Wireless Networks for the Internet of Things," *IEEE Access*, vol. 5, pp. 7097–7114, 2017.

[15] "Near Field Communication - Interface and Protocol (NFCIP-1)," ECMA, Tech. Rep., June 2013.