Authentication and authorization for interoperable IoT architectures

Nikos Fotiou and George C. Polyzos

Mobile Multimedia Laboratory, Department of Informatics
School of Information Sciences and Technology
Athens University of Economics and Business
Evelpidon 47A, 113 62 Athens, Greece,
{fotiou,polyzos}@aueb.gr

Abstract. Advances in technology have enabled the creation of "smart" Things, fostering the vision of the Internet of Things (IoT). Smart Things have connection capabilities, they support Internet protocols and they even come with operating systems and Application Programming Interfaces. The pursuit for a protocol stack that will support the IoT has resulted, so far, in an ecosystem of heterogeneous and non-compatible solutions that satisfy the requirements of particular vertical sectors ("silos"). For this reason, several research initiatives, driven by both academia and industry, investigate the potential of an interoperable IoT architecture, i.e., an architecture that will provide a common and horizontal communication abstraction, which will act as interconnection layer among all prominent IoT protocols and systems. Securing such an architecture, which includes many stakeholders with diverse interests and security requirements, is not a trivial task. In this paper, we present an authentication and authorization solution that facilitates the interoperability of existing IoT systems. This solution achieves endpoint authentication, encryption key establishment, and enables third parties to define fine-grained, domain-specific access control policies. Things store minimal information, perform only ultra-lightweight computations, and are oblivious about the business logic and processes involved in the authentication and authorization procedures. Furthermore, the proposed solution preserves end-user privacy and can be easily incorporated into existing systems.

1 Introduction

Smart Things support a wide range of connectivity options (for example 6LoWPan, ZigBee, Bluetooth Low Energy, Ethernet), Inter-networking protocols (like MQTT, CoAP), and even lightweight operating systems (for example RIoT and Contiki). However, the arms race for an "Internet of Things" (IoT) architecture, has resulted in numerous, diverse, and competing systems that often satisfy only the requirements of a particular vertical use case. However, an "interoperable" IoT architecture would

provide significant advantages to society (and data producers or owners), allowing (controlled) data use from all domains by all applications–through silo boundaries.[1] Several research efforts, driven by academia and industry, have sprung up investigating this potential.

Securing smart Things-based systems is a challenging task by itself, therefore it comes as no surprise that the security of an interoperable IoT architecture is a problem that cannot be easily addressed. An interoperable IoT architecture should interconnect various stakeholders with diverse security needs, requirements, and capabilities, it should allow for flexible user identities and generic access control policies, and it should enable federation of security providers while facilitating at the same time compartmentation and isolation of sensitive business processes.

In this paper, we build on our previous work [9] and we present the design, implementation, and evaluation of a security solution that achieves endpoint authentication, encryption key establishment, and access control delegation. The proposed solution, which was tailored to the IoT and is appropriate for constrained devices, enables the interoperability among various stakeholders without sacrificing end-user privacy and security. With our solution, user-related information never leaves user management systems, while service providers can easily secure their offerings.

In this paper we make the following contributions:

- We improve the performance and the security of our original design [9]. In particular with the new design Things do not maintain any state for unauthenticated connections. Furthermore, the new design is compatible with the (D)TLS handshake with pre-shared keys, hence significant security properties can be claimed.
- We extend [9] to support multi-stakeholder IoT-based services, facilitating this way interoperability.
- We implement our solution and we integrate it in the INTER-IoT interoperable gateway [4], enabling its use from all INTER-IoT systems and applications/use-cases.

Even though there are many somewhat similar access control solutions for Internet applications, our solution has been designed for and tailored to the IoT and has many significant advantages in this domain. Furthermore, compared to existing solutions for the IoT, and as discussed later in Section 6, our solution is more lightweight, it has better security and privacy properties, it does not require from Things to be online, and completely hides business logic, semantics, and processes from the Things and the end-users. The latter property is of particular importance when it comes to interoperability and business-to-business services. Finally, as we detail in Section 3.3, our solution can be integrated into the (D)TLS handshake–(Datagram) Transport Layer Security–hence existing (D)TLS-based applications can benefit from our approach without any modification.

The remainder of this paper is structured as follows. In Section 2 we give an overview of our system and we present an illustrative use case. In Section 3 we detail the design of our solution and its integration with

[1] Data (and information) is not only a non-rivalrous good, it is anti-rivalrous [1], providing (potentially) more value the more it is used.

(D)TLS. In Section 4 we evaluate the qualitative and security properties of our solution. In Section 5 we discuss the integration of our solution with an existing interoperable IoT platform. We compare our solution with existing related work in Section 6 and we provide our conclusions in Section 7.

2 System Overview

In order to give a better overview of our system we present the use case of a "smart port" (similar to a use case of the INTER-IoT project [4] not entirely by coincidence). In this use case, illustrated in Figure 1, port employees want to access resources provided by Things embedded in containers arriving at the port. Container owners want to make sure that these resources can be accessed only by the port employees. On the other hand, the port authority does not want to allow third parties to access its user management system.

Using our solution this problem can be overcome as follows. The port authority extends its user management system to support access control policies, as well as our protocol. Then, it creates an access control policy that defines who are the port employees and assigns to this policy a URI. From a high-level perspective, the user management system of the port authority can now be viewed as an RPC server: whenever a port employee makes a call to the policy URI, using as input call parameters a "token" and his identification data, the server generates and responds back with an encryption key. A policy URI can then be used by container owners to protect their devices (Steps 1-3).

Each container owner "registers" its devices to the user management system of the port authority and receives back a secret key which is installed in the devices, along with the policy URI (steps 4-6). Suppose now that a port employee wants to access some information provided by a protected device. Initially, he sends an "unauthorized request" and receives back a token and the URI of the policy generated during step 1 (steps 7,8). Then he performs an RPC call to the user management system and obtains an encryption key (steps 9,10). With our solution, the device can also calculate the same encryption key, offline, without any communication with the user management system. Since both entities, i.e, the port employee and the Thing, now share the same key, they can use it for securely exchanging data using a protocol such as (D)TLS with pre-shared keys (step 11). The encryption key generation process guarantees user authentication and authorization, as well as device authentication.

Our system achieves the following goals:

- **Transparency.** Container owners are oblivious about the implementation, structure, and content of the port authority user management system. The application implemented in the Things, does not contain any port authority specific logic.
- **Flexible security management.** The port authority can modify the access control policy stored in its user management system without needing to update policy URIs or notify container owners.

– **User privacy preservation.** Things learn no user-specific information. They only information they can deduce is that a user has business relationships with a specific user management system.
– **Lightweight security.** Things have to maintain only a secret key and a policy URI per resource. Furthermore, Things do not have to be connected to the Internet, or to any other network.

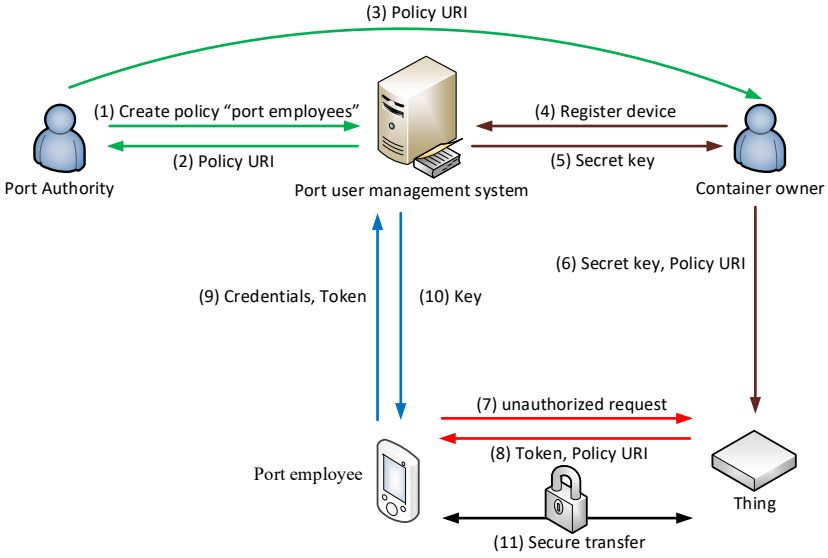

Fig. 1. Smart port use case.

3 System Design

3.1 Preliminaries and notation

Our construction relies on a keyed-hash message authentication code (HMAC). We refer to the digest of a message m using an HMAC function h and a key k as $h_k(m)$. Moreover, we refer to the concatenation of n messages as $m_1||m_2||...||m_n$. Entities in our system are uniquely identified either by an identifier or a URI. We refer to the identifier of an entity A as ID_A. Similarly, we refer to the URI of entity B as URI_B.
The core entity of our system is the *Access Control Provider* (ACP) (i.e., the "enhanced" user management system of our use case). An ACP implements the following algorithms:
– **storePolicy**(policy): Stores an access control policy and returns a policy unique URI_{policy}.

- **register**(URI_{policy}, $URI_{resource}$): Registers an IoT resource, protected using URI_{policy}, and returns a secret key (sk). This secret key is $URI_{resource}$ specific. An ACP should check if the entity that invokes this algorithm is the legitimate owner of $URI_{resource}$. However, this process is out of the scope of our work.
- **authorize**(identificationData, URI_{policy}, token, $URI_{resource}$): Examines if a user identified by identificationData can be authorized using URI_{policy}. If this is true, it generates a user specific ID_{user} and invokes the keyGen() algorithm described below. The ID_{user} generation process is ACP specific. Every time the same user invokes the $authorize()$ algorithm the ID_{user} may be the same (although this enables user tracking): the only requirement imposed by our system is that it must not be possible for two distinct users to receive the same ID_{user}.
- **keyGen**(ID_{user}, URI_{policy}, token): Creates an ephemeral encryption key by calculating $h_{sk}(ID_{user}||URI_{policy}||token)$, where sk is the key generated using the $register$ algorithm and $token$ is a session specific random number generated by the protected device. It outputs ID_{user} and the ephemeral encryption key.

Each Thing implements the $keyGen()$ algorithm as well. Users have some short of business relationship with an ACP. We assume that users can securely communicate with their ACP and ACPs implement a secure method for authenticating users. Finally, users and Things can communicate with each other. We make no security assumption about the latter communication channel, i.e., any third party can monitor and tamper with the messages exchanged between a user and a Thing.
In addition to attackers monitoring the communication channel between users and Things, our threat model assumes unauthorized users trying to get access to a protected resource, as well as malicious devices trying to impersonate legitimate Things.

3.2 Protocols

Our solution is composed of the following protocols: *Setup*, *Unauthorized request*, *User authentication and authorization*, *Authorized request.*

Setup: The goal of the setup protocol is to enable resource owners to "pair" devices that provide a protected resource with one or more ACPs. Every *resource owner* that wants protect a resource $URI_{resource}$ using an access control policy URI_{policy}, invokes over a *secure communication channel* the $register()$ algorithm, receives a secret key sk, and installs it in the corresponding Things. It should be noted that resource owners, Things, and end users do not have to be aware about the implementation details and the business semantics and logic of an access control policy: the only information they learn about a policy is its URI_{policy}. A resource owner may register a resource with multiple ACPs. In that case it will configure the Thing with all URI_{policy} and the corresponding secret keys. Furthermore, sks are only used for generating other keys and they are never communicated to other entities.

Unauthorized request: The goal of the unauthorized request protocol is to provide users with the necessary authentication and authorization information. A user wishing to access a protected $URI_{resource}$ initially sends to the Thing an *unauthorized* request over an unprotected communication channel. Then, the Thing responds with a *token* and a list of URI_{policy}. A token is a public, random variable unique among all sessions of that specific Thing. The protocol used for making these requests is application specific, e.g., a user may request a resource over HTTP, CoAP [15], or any other protocol.

User authentication and authorization: With this protocol, users authenticate themselves to an ACP and receive an ephemeral encryption key. Upon receiving a response to an unauthorized request, the user selects a suitable URI_{policy} and invokes the *authorize*() algorithm over a *secure communication channel*. If the user can be authorized for URI_{policy}, the ACP invokes the *keyGen*() algorithm and sends back to the user, the ID_{user} and the ephemeral encryption key, over the same secure communication channel. A Thing can also calculate the same ephemeral key, offline, using the *keyGen*() algorithm. However, no third party, including the user, can calculate this key since the secret key used by the HMAC calculation is only known to the ACP and the Thing. It is reminded that the latter secret key is $URI_{resource}$ specific.

Authorized request: The goal of this protocol is to enable Things to generate offline the ephemeral key that an authorized user received from the ACP, as well as to provide means for using this key for securing subsequent communication. In order for the Thing to generate the ephemeral key (i.e., invoke the *keyGen*() algorithm) it needs to learn (i) the token it generated during the unauthorized request, (ii) the selected URI_{policy}, and (iii) the ID_{user}. All these can be provided by the user over an unprotected communication channel. Using this key to protect subsequent communication cannot be trivially achieved in a secure way. For this reason, we rely on (D)TLS–although other protocols can be considered as well.
Figure 2 updates Figure 1 with the defined algorithms, protocols, and entities. The authorized request protocol is not illustrates since for that we consider (D)TLS. In the following we present the integration of our solution with (D)TLS with pre-shared keys.

3.3 (D)TLS integration

The goal of (D)TLS is to allow two communicating endpoints, a client and a server, to establish a secure communication channel by executing a "handshake" protocol over an unprotected channel [13]. The security of this protocol can be based on public-key cryptography or on a pre-shared secret key [7] (or in a combination of these two approaches). (D)TLS with pre-shared secret key (PSK-(D)TLS) is ideal for constrained devices since it can be implemented using only a few, lightweight operations. With

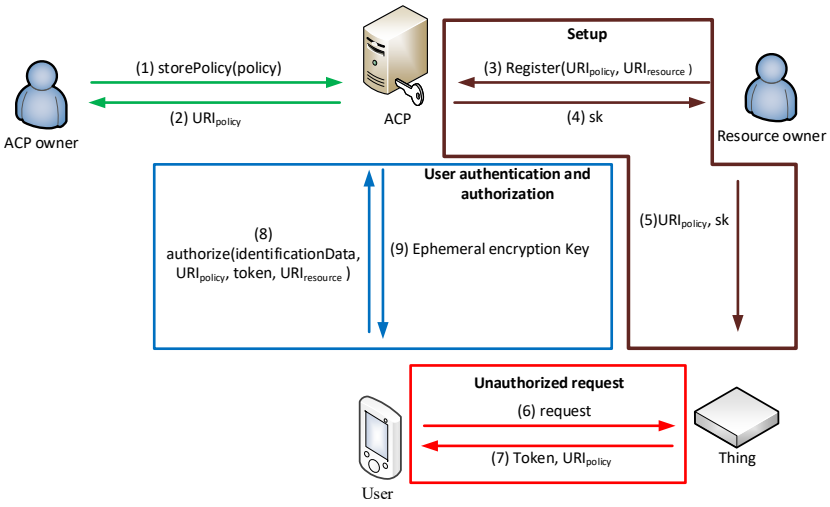

Fig. 2. Algorithms, protocols, and entities of the proposed solution.

PSK-(D)TLS the communicating endpoints use their pre-shared secret key to derive a "pre-master secret key," and then they use this key as input to a key derivation function (KDF) to calculate a "master secret key". The KDF, in addition to the pre-master secret key, uses as input two random numbers generated by the communicating endpoints and exchanged using the handshake protocol.

In a nutshell (and from a really high perspective), in order to collect the KDF input parameters, a client and a server exchange two "Hello" messages, that include the random numbers selected by each endpoint, and two "KeyExchange" messages that include auxilliary information. Our goal is to use the ephemeral encryption key produced by our solutions as the (D)TLS pre-shared secret key. In particular we implement the unauthorized request and the authorized request protocols using the (D)TLS handshake messages.

The unauthorized request protocol requires from a Thing (server) to send to a user a token and a list of URI_{policy}. This information can be encoded in the "psk_identity_hint" field of the server KeyExchange handshake message. This field is a byte array of size up to 2^{16} bytes and it is used by the server "[...] to help the client in selecting which identity to use." Furthermore, the authorized request requires from a user to send to a Thing the selected URI_{policy}, the generated ID_{user}, and the token. This information can be encoded in the "psk_identity" field of the client KeyExchange handshake message. This field is a byte array of size up to 2^{16} bytes and it is used by the client to " [...] indicate (to the server) which key to use." Hence, not only we can transfer our protocol parameters using the (D)TLS handshake, but also, we do not violate the semantics of the used fields.

The integrated procedure is illustrated in Figure 3. In the example depicted in this figure, a user (acting as (D)TLS client) wishes to access a protected resource provided by a Thing (acting as the (D)TLS server). The user initiates the communication by sending a client "Hello" handshake message. The Thing responds with a server "Hello" followed by a server "KeyExchange" handshake message. The latter message includes a token and a list of URI_{policy}, encoded in the "psk_identity_hint" field. The Thing selects an appropriate ACP, executes the *user authentication and authorization* protocol and receives the ephemeral encryption key and its ID_{user}. Then the user sends a client "KeyExchange" handshake message and includes the token, the selected URI_{policy}, and the ID_{user} in the "psk_identity" field. With the reception of the latter message, the Thing can invoke the *keyGen*() algorithm and generate the ephemeral encryption key. As a final step, both endpoints execute the (D)TLS KDF and generate the master secret key. From this point on, all subsequent messages can be secured this key.

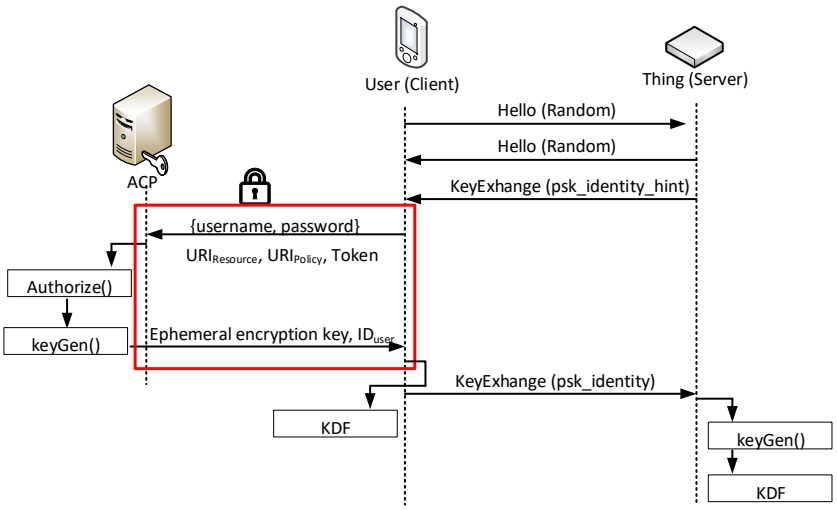

Fig. 3. Integration with (D)TLS.

4 Evaluation

In the following we evaluate our solution. We consider the (D)TLS integrated version. Our solution has the following properties:

It facilitates interoperability. Our solution hides the business logic and semantics of each stakeholder. It defines a simple API that allows

applications to interact with ACPs and at the same time it gives great flexibility on how an ACP is implemented. ACPs enable compartmentation and isolation of sensitive business processes allowing stakeholders to modify their (security) policies without affecting the applications that are using them. Business-to-Business services can be easily implemented: a company A can offer services to the users of a company B simply by leveraging a URI_{policy} provided by company B.

It facilitates application development. By integrating our solution with (D)TLS we provide a straightforward and transparent way for application developers to include it in their products. As a matter of fact, by incorporating our protocol into the DTLS implementation of the BouncyCastle library[2], and by using a single ACP and hardcoded $identificationData$ it was possible to port existing (example) applications without any modification.

It is lightweight and it protects users' privacy. With our solution, Things have only to perform a single HMAC calculation, in addition to the operations required by (D)TLS. Furthermore, Things do not have to be connected to the Internet and do not have to maintain any state for unauthorized requests. Finally, Things learn no user specific information apart from the ID_{client}.

4.1 Security evaluation

Providing that two users (clients) do not share the same pre-shared key and providing that pre-shared keys have enough entropy, PSK-(D)TLS has the following security properties:

- **Communication integrity.** The PSK-(D)TLS handshake protocol makes sure that any modification to the exchanged messages can be detected (but it cannot be prevented).
- **Confidentiality.** The master secret key cannot be guessed and it can be used to protect the integrity and the confidentiality of the messages exchanged after the completion of the handshake.
- **Server authentication.** The PSK-(D)TLS handshake protocol makes sure that man in the middle attacks can be detected (but not prevented). Hence it is not possible for an attacker to impersonate a server.

The pre-shared key used in our solution is the ephemeral encryption key generated by the ACP.

Theorem 1. *Two users identified by different identificationData cannot obtain the same ephemeral key.*

Proof. The *keyGen*() algorithm of an ACP, i.e., the algorithm that generates the ephemeral key, uses as input the variable ID_{user}. The latter variable is produced by the *authorize*() algorithm. Given two users identified by $identificationData$ A and B respectively, then by definition

[2] https://www.bouncycastle.org/

$authorize(A) \neq authorize(B)$.

PSK-(D)TLS in its simplest form (i.e., without using the Diffie-Hellman key exchange algorithm) does not provide forward secrecy.
An additional security feature of our solution is that it can immediately prevent users with revoked access rights to retrieve a resource. This is achieved by having Things generating a token in every session and by keeping track of the already used tokens: fresh tokens force users to communicate with an ACP in order to retrieve a new ephemeral encryption key. That way, revoked users can be quickly blocked from obtaining such a key.
The security properties of our solution depend on the secrecy of the secret key installed in Things by the resource owners. If this key is breached, then the *setup* protocol should be re-executed for all Things sharing the same key. Nevertheless, no further update is required (e.g., user applications do not have to be modified)

5 Integration with an existing interoperable platform

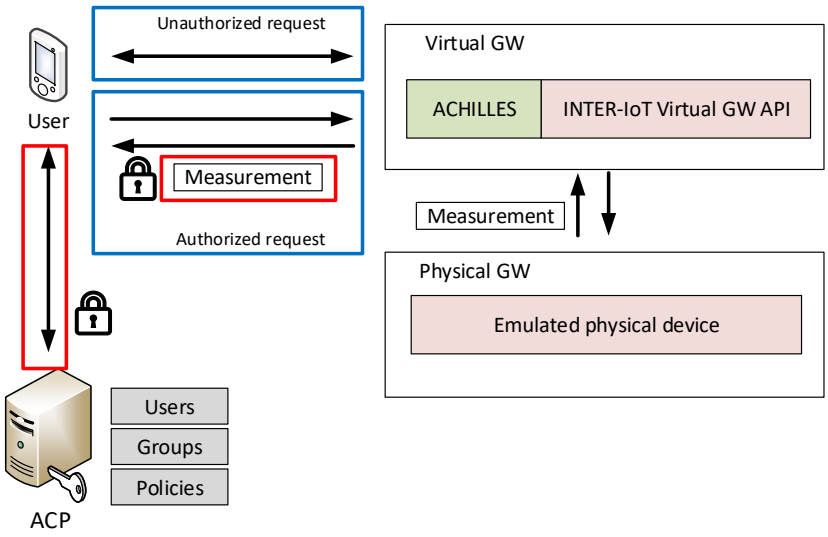

Fig. 4. Integration with the INTER-IoT architecture.

As proof of concept, we implemented our solution for the INTER-IoT interoperable IoT gateway. INTER-IoT gateway is a modular system that

targets to join various IoT platforms and technologies under a common API. The INTER-IoT gateway is composed of two parts: the virtual part and the physical part. The physical part provides an abstraction that can be used for interacting seamlessly with Things using various link layer technologies and protocols, whereas the virtual part exposes an API that can be invoked by a user. An API call to the virtual part can be translated into a call to the physical part or it can be translated and forwarded to another IoT platform. If necessary, the user will receive the appropriate response.
In order to accommodate various technologies, the virtual part of the gateway supports plug-in extensions. We implemented our solution as a plug-in, code-named ACHILLES (stands for Access Control and autHenticatIon deLegation for interoperabLE IoT applications). Our plug-in extends the API of the virtual part of the gateway and implements the protocols described in the previous section.
However, the INTER-IoT API is implemented over HTTP(s), hence our solution had to be implemented at the layer. For this reason, we implemented the TLS KDF in the application layer and we used HTTP headers in order to exchange the necessary information. The derived master secret key is used to encrypt measurements provided by an emulated IoT device. The encrypted measurements are then included in the payload of the HTTP response, sent from the gateway to the user.
As an ACP we used a custom-made user management system. In this system users are identified by a pair of a username and a password. Simple access control policies can be defined using NIST's core Role-based Access Control (core-RBAC) model [8], i.e., the ACP owner creates users, organizes users in groups, and defines access control policies based on these groups. Furthermore, users can communicate with the ACP and exchange data securely using HTTPS.

6 Related work

The key characteristics of our solution are the following: it is lightweight, it preserves user privacy, it can be used even if Things are not connected to the Internet, Things and applications are business process agnostic, and it is general purpose.
Various systems try to implement Role-Based Access Control (RBAC), or Attribute Based Access Control (ABAC), either by storing user credentials in the Thing or by using a federated identity system, such as OAuth [11] (e.g., as used in [6]) or OpenID [12] (for example, as described in [5]). Storing access control policies in Things raises many scalability and security concerns. For example, updating an access control policy requires communication with all involved Things, whereas with our system, policy modifications take place only on ACPs. Of course, storing user management related information in Things creates many privacy riks. Furthermore, federated identity systems, such as OpenID and OAuth require digital signature verification, which might be too computationally heavy for many IoT devices or applications.
The disadvantages of RBAC/ABAC systems can be overcome by using capabilities tokens. A capabilities token defines the operations that a user

is authorized to perform over an object. Capabilities tokens are issued and digitally signed by a third trusted entity. Capabilities-based access control (CBAC) has been studied in the context of the IoT by many research efforts (for example [10], [14]). The main drawback of these systems is that Things have to understand the business logic encoded in a token. With our solution business logic and semantics are transparent to Things and to users.

Eclipse Keti [3] is a token-based access control system which hides business logic from Things. Using Keti, a Thing–or any application–may query an "access control service" if a user is allowed to perform a particular operation. The main drawback of Keti is that it requires Things to be able to communicate with access control services. With our solution Things can be isolated from the rest of the world. Furthermore, using Keti, access control services should be aware of the possible operations that can be executed in a Thing. This creates scalability and privacy issues. Using our system, ACPs (i.e., the entity that holds the same role as the access control service) does not learn the operations that a user wants to perform; it is even possible to hide from an ACP the fact that a user interacts with a Thing. Finally, with our system it is possible to create re-usable policies. For example, the policy "port employees" defined in our use case in Section 2 can be used by many systems (not necessarily IoT specific).

Musquitto auth-plug [2] is a plug-in used for authorizing Musquitto MQTT broker users. Musquitto auth-plug can be configured to work with multiple and diverse user management systems and, similarly to our solution, it can be used as a security add-on to an existing deployment. However, this plug-in is product and protocol specific. Furthermore, it operates in a way similar to RBAC/ABAC systems.

We see, therefore, that our proposed approach, while it seems similar to many existing and proposed solutions, is significantly different, designed specifically for the IoT, and very flexible, allowing its effortless integration with diverse existing business systems and applications, or its incorporation into new IoT system or application designs.

7 Conclusions

In this paper we presented a security solution for an interoperable IoT architecture. The proposed solution achieves endpoint authentication, user authorization, and key establishment between two endpoints. The proposed solution relies on a third party referred to as the Access Control Provider (ACP). The ACP, which can be implemented alongside the user management system of a company, guarantees that no user specific information is stored in Things. Moreover, the ACP allows users to securely communicate with Things without any pre-established secret information. By storing access control policies in ACPs, our solution facilitates security management, since a modification to an access control policy does not have to be propagated to the involved Things. Moreover, by hiding business logic from Things, Business-to-Business services are facilitated.

The proposed solution can be seamlessly integrated with (D)TLS. In particular, we leveraged specific fields of the (D)TLS handshake protocol (without breaking their semantics) to transfer our protocol specific parameters. Then we used the (D)TLS key derivation function to securely create an encryption key that can be used for protecting the integrity and the confidentiality of all subsequent messages. This integration is important for many reasons: it facilitates application development, since existing applications based on (D)TLS can be easily ported to our system, it enhances the security of our approach, and it provides a mechanism for (D)TLS to create pre-shared keys.
As a proof of concept, we incorporated our solution to the INTER-IoT interoperable gateway and we extended its API. Our extensions allow INTER-IoT gateway-based systems to include existing user management systems with very little effort. Given the security requirements of the scenarios considered by the INTER-IoT team, this is an important development since involved stakeholders do not have to allow third parties to access their (critical) security systems.

Acknowledgment

This work was funded through INTER-IoT Collaboration Agreement #52 (ACHILLES), which is administered through AUEB-RC. INTER-IoT has received funding from the EC through programme H2020. The paper presents the views of the authors and not necessarily those of the EC or the INTER-IoT consortium

References

1. Anti-Rivalry definition. URL https://wiki.p2pfoundation.net/Anti-Rivalry. (last accessed 8 Jul. 2018)
2. Authentication plugin for Mosquitto with multiple back-ends. URL https://github.com/jpmens/mosquitto-auth-plug. (last accessed 8 Jul. 2018)
3. Eclipse Keti. URL https://projects.eclipse.org/proposals/eclipse-keti. (last accessed 8 Jul. 2018)
4. INTER-IoT project home page. URL http://www.inter-iot-project.eu. (last accessed 8 Jul. 2018)
5. Blazquez, A., Tsiatsis, V., Vandikas, K.: Performance evaluation of openid connect for an iot information marketplace. In: 2015 IEEE 81st Vehicular Technology Conference (VTC Spring), pp. 1–6 (2015)
6. Cirani, S., Picone, M., Gonizzi, P., Veltri, L., Ferrari, G.: IoT-OAS: An OAuth-based authorization service architecture for secure services in IoT scenarios. IEEE Sensors Journal **15**(2), 1224–1234 (2015)
7. Eronen, P., Tschofenig, H.: Pre-shared key ciphersuites for transport layer security (TLS). RFC 4729, IETF (2005)
8. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. **4**(3), 224–274 (2001)

9. Fotiou, N., Kotsonis, T., Marias, G.F., Polyzos, G.C.: Access control for the internet of things. In: 2016 ESORICS International Workshop on Secure Internet of Things (SIoT), pp. 29–38 (2016)
10. Gusmeroli, S., Piccione, S., Rotondi, D.: A capability-based security approach to manage access control in the internet of things. Mathematical and Computer Modelling **58**(5), 1189 – 1205 (2013). The Measurement of Undesirable Outputs: Models Development and Empirical Analyses and Advances in mobile, ubiquitous and cognitive computing
11. Hardt (ed.), D.: The OAuth 2.0 authorization framework. RFC 6749, IETF (2012)
12. Recordon, D., Reed, D.: OpenID 2.0: a platform for user-centric identity management. In: Proceedings of the second ACM workshop on Digital Identity Management, DIM '06, pp. 11–16. New York, NY, USA (2006)
13. Rescorla, E., Modadugu, N.: Datagram transport layer security version 1.2. RFC 6347, IETF (2012)
14. Seitz, L., Selander, G., Gehrmann, C.: Authorization framework for the internet-of-things. In: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a, pp. 1–6. IEEE Computer Society, Los Alamitos, CA, USA (2013)
15. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252, IETF (2014)