Access Control for the Internet of Things

Nikos Fotiou, Theodore Kotsonis, Giannis F. Marias, 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
Email:{fotiou,kotsonis,marias,polyzos}@aueb.gr

***Abstract*—As we are moving from networked "Things" towards the Internet of Things (IoT), new security requirements arise. Access control in this new environment is a burgeoning and challenging problem. On the one hand, an access control system should be generic enough to cover the requirements of all the new exciting applications that become pervasive with the IoT. On the other hand, an access control system should be lightweight and easily implementable, considering at the same time the restrictions that Things impose. In this paper, we develop an access control system which enables offloading of complex access control decisions to third, trusted parties. Our system provides Thing authentication without public keys and establishes a shared symmetric encryption key that can be used to secure the communication between authorized users and Things. Our design imposes minimal overhead and it is based on a simple communication protocol. The resulting system is secure, enhances end-user privacy and the architecture facilitates the creation of new applications.**

I. Introduction

Nowadays, "smart" Things have become an integral component of many systems. Industrial production, health services, traffic monitoring, and many other fields have been greatly improved by the widespread adoption of sensors, actuators, embedded systems, identification technologies, and mobile devices. With the pervasiveness of and advances in networking technologies and the mass production of smart(er) Things, we are on the verge of moving from vertical application silos towards an Internet of Things (IoT). The IoT will enable the creation of general purpose applications that will harvest the full potential of the unique features of smart Things.

This vision of the IoT raises significant security and privacy concerns. With applications impacting not only the virtual world, but now directly the real world, concerns are pervasive and justified. Things can be intrusive, have access to sensitive information, can be easily tampered with and at the same time Things might not have enough computational, storage, or energy capacity, and in general might have significant limitations.

In this paper we discuss the problem of access control for the IoT. Access control is an essential component of many applications since it prevents unauthorized access to information (and/or activation). Access control is expected to become even more critical in the IoT, but at the same time even more challenging. Access control in the IoT poses some unique requirements. The devices in which protected resources are stored (i.e., the Things), are usually small devices with limited storage capacity, power, and processing capabilities, in order to be inexpensive. In many cases Things are "exposed" to tampering, whereas in many application scenarios, after Things are deployed, it is not easy to access them remotely. Things usually are not able to perform "heavy" tasks, such as complex cryptographic operations. Storing user credentials or any other sensitive information in a Thing is undesirable both from a security and a storage overhead perspective. Moreover, users are not expected to be willing to share sensitive information with a Thing (such as credentials or information about their role in a company), even if this information is required by an access control mechanism.

In this paper we propose an access control system that overcomes these challenges. In a nutshell, our system secures resources provided by Things by using "pointers" to access control policies. Access control policies are stored in trusted third parties, referred to as *Access Control Providers* (ACPs). ACPs are responsible for evaluating users against an access control policy and communicate the outcome of this evaluation back to the Thing. We design a secure, generic and lightweight communication protocol, which preserves user privacy and facilitates application development. The work reported in this paper extends our previous work published in [1][2] by adapting it in the context of the IoT. In particular, our previous work assumes a secure communication channel between a client and a server (the Thing in this paper). This channel, which is implemented using public-key cryptography, assures end-point identification, as well as, the integrity and the confidentiality of the exchanged messages. But, when it comes to the IoT, such a communication channel cannot be trivially constructed, therefore it cannot be simply "assumed" by an access control solution. The system proposed in this paper provides Thing authentication without public keys, and enables the establishment of a shared symmetric encryption key which can be used to secure the communication between authorized users and Things. Moreover, in this paper a simpler communication protocol is used, that requires less storage and computation overhead.

The remainder of this paper is organized as follows. In Section II we discuss related work in the area. In Section III we present the design of our system and in Section IV we evaluate it. In Section V we discuss some details of our system, a possibly extension, as well as, our system's business perspectives. Finally, in Section VI we present our conclusions and plans for future work.

II. Related work

DCAF [3] (Delegated CoAP Authentication and Authorization Framework) is an IETF Internet Draft with very similar goals. DCAF defines a protocol for delegating client authentication and authorization in constrained environments. A Thing can use this protocol to delegate authentication of clients and management of authorization information to a trusted, more powerful, Server Authorization Manager (SAM). A SAM authenticates clients and creates symmetric encryption keys which are then delivered both to (authorized) clients and to Things. These keys are used for establishing a Datagram Transport Layer Security (DTLS) channel between the client and the Thing. In DCAF, a Thing and a SAM should belong to the same administrative domain, they should share a secret, they should have loosely synchronized clocks, and there should always be a communication channel between them. In our solution, ACPs (which implement functions similar to SAMs) can be independent entities, they do not have to be synchronized with the Things, neither do they have to communicate directly with them. As a matter of fact, in our system an ACP does not have to know that it performs a user authentication and authorization on behalf of a Thing. In DCAF when a user requests authentication and authorization from a SAM, he should include in his request the resource in which he is interested. In our system, users include in their authentication/authorization messages a Thing identifier, therefore, an ACP learns less fine grained information about the user.

A common method for implementing an access control system is by defining the *roles* (or *attributes*) that a user should have in order to access a resource. This method is commonly referred to as Role-Based Access Control (RBAC) [4] (or Attribute Based Access Control (ABAC) when attributes are used instead of roles [5]). RBAC has been explored in the context of the IoT (e.g., in [6]). In this context, access control policies are defined directly in a Thing, or in a centralized gateway which acts as an interface to Things. User authentication is performed either by storing user credentials in the Thing/gateway or by using a federated identity system, such as OAuth [7] (e.g., as used in [8]) or OpenID [9] (e.g., as described in [10]). Storing access control policies in Things (or even in gateways) has many scalability and security issues: policy management becomes difficult, the access control system cannot be easily re-used by diverse applications, support for users belonging to different administrative domains is hard, and user privacy can be jeopardized.

Another approach for implementing access control is by using the so-called *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 (e.g., [11], [12], [13]). CBAC is more lightweight, more scalable and more secure compared to RBAC/ABAC. Our solution resembles to a CBAC system, however, it has a key difference: in our system the trusted party that authenticates users decides if they abide by an access control policy and responds with (roughly) a 'yes' or 'no,' rather than authenticating the users and responding with their capabilities. With our system Things do not have to be able to interpret complex token formats. Moreover, in our system trusted parties do not have to be aware of the services a user wants to access (in contrast to CBAC where the trusted party needs to know what a user wants to access in order to create the appropriate capabilities token).

Our work defines its own protocol for establishing a shared symmetric encryption key. Other similar protocols (e.g., [14] and its improvements) could be used instead (with some modifications). The main difference of our protocol compared to existing work is that it allows the Thing to calculate the shared symmetric key by itself, whereas in most of the existing protocols the symmetric key is transmitted encrypted by the other communicating party. This property requires simpler operations (hashing vs. decryption) and fewer messages.

III. System design

A. System entities

Our system considers the following entities: *Things*, resource *owners*, resource *clients*, and *access control providers (ACPs)*. The goal of a resource owner is to provide a *resource* only to *authorized* clients, using a Thing. Access to a resource is regulated by an *access control policy*. Each access control policy is stored in an ACP and maps the *identity* of a client to a Boolean output (true, false). Therefore, clients should have some form of *business relationship* with the ACP. When the output of an access control policy is true, the client is considered authorized. Ideally, an access control policy should not be Thing/resource specific, e.g., an access control policy may output "true" for any customer of a particular company, this way access control policies become re-usable.

Although by design ACPs and resource owners are two distinct entities, in reality there can be cases where these roles are held by the same real world entity.

B. Design Goals

Our system design is driven by the following goals:

- The designed system should not rely on client sensitive information stored in a Thing. Storing such information in a Thing introduces security and privacy risks, as well as, management overhead (e.g., consider a scenario where a resource is offered by hundreds of Things).
- The Thing part of our system should be lightweight and should introduce as little overhead as possible. The designed system should not require from Things to implement computationally intensive tasks (e.g., use public-key based encryption).
- End-user privacy should be protected. Clients should not have to reveal any sensitive information to Things.
- The designed system should be easily manageable: policy modifications and user addition/removal should involve as little interaction with the Things as possible.
- The designed system should be generic and not designed with the requirements of a specific application in mind–or even of a specific application domain (silo).

C. System setup

Our system assumes an out-of-band and secured setup phase. During this phase each ACP generates and securely stores a Master Secret Key (MSK). In addition, access control policies are created and stored in ACPs. For each policy a *Uniform Resource Identifier* (URI) is generated. These URIs are of the form "*ACP location/access control policy name*". A policy URI, henceforth denoted as URI_{policy}, may be used by many resource owners: a resource owner does not have to be aware of the rules and the implementation details of an access control policy; the only information that a resource owner needs in order to protect a resource is a URI_{policy}. Every resource owner that wants to use a policy to protect a resource stored in a Thing identified by ID_{Thing}, issues a *secret key request* to the appropriate ACP. The ACP verifies that the resource owner is allowed to issue a secret key request for that particular Thing identifier (see Section V-A for a discussion on that) and uses a secure HMAC and the MSK to calculate a secret key as follows:

$$SK_{acp,Thing} = HMAC_{MSK}(ID_{Thing}) \quad (1)$$

The calculated SK is then securely delivered to the resource owner.

Each resource owner configures Things with the appropriate URI_{policy} and $SK_{acp,Thing}$. For this reason, each Thing maintains an *Access Table* that contains tuples of the form $[Resouce, Policy, SK]$, where *Resource* is the name of the protected resource, *Policy* is the URI of the policy that protects the resource, and *SK* is the secret key that has been generated by the ACP and installed by the resource owner during the setup phase. Moreover, resource owners configure client software with a mechanism that detects legitimate Thing identifiers (e.g., with a list of valid Identifiers, with a prefix that a Thing identifier should have, etc.).

Finally, each client receives from the ACP(s) with which he has business relationship a unique, public, identifier. The semantics of this identifier, henceforth denoted as ID_{client}, are ACP specific. This identifier should be treated by a 3^{rd} party (including Things) as an arbitrary number.

Figure 1 illustrates the system setup phase (ACP's MSK generation is omitted). In the illustrated example an access control policy (Policy1) is installed in an ACP (A1) and a resource owner learns the URI of that policy ($URI_{policy1}$). Then, the resource owner requests a secret key for a Thing (T1) from A1, A1 generates this key by using its MSK (MSK_{A1}), and sends it back to the resource owner. Finally, the resource owner configures T1 with a resource (R1), the corresponding policy URI, and the appropriate secret key (i.e.,$URI_{policy1}$, $HMAC_{MSK_{A1}}(T1)$). The Access Table of T1 is modified accordingly. It should be noted here that this phase takes place out-of-band and all operations are secured.

D. Unauthorized request

A client wishing to access a $resource$ stored in a ID_{Thing}, initially sends an unauthorized request. The client learns the ID_{Thing} either by having it pre-configured in his software, or by using a service discovery mechanism. An unauthorized request is transmitted unprotected and includes the ID_{client}. Upon receiving an unauthorized request, a Thing retrieves URI_{policy} and $SK_{acp,Thing}$ from the Access Table and generates a *token t*. A token is a (public) variable unique among all sessions of that specific Thing.[1] Then, the Thing uses $SK_{acp,Thing}$ and calculates a *session key* as follows:

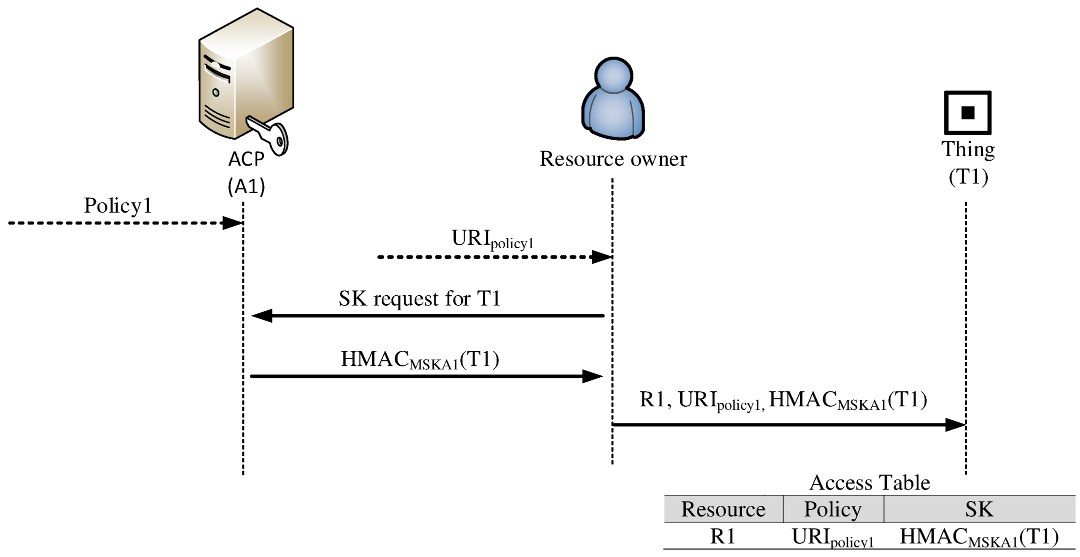
$$SK_{session} = HMAC_{SK_{acp,Thing}}(URI_{policy}, t, ID_{client}) \quad (2)$$

All tokens and session keys are stored in a *Token Table* that contains tuples of the form $[SessionId, Token, Expires, Resource, SK]$, where *SessionId* is an identifier for that session (e.g., client's IP address and port), *Expires* is the expiration time of the token, *Resource* is the resource requested by the client, and *SK* is the output of equation 2. Finally, the Thing sends the URI_{policy} and the token back to the client. This message is also transmitted unprotected

E. Client authentication and authorized request

Upon receiving the Thing's response, a client sends an authentication request to the appropriate ACP, which is found using the URI_{policy} included in the received message. The semantics of this request, which is transmitted over a secure communication channel, are ACP specific. This request must contain the ID_{Thing}, the URI_{policy}, and the token t, received by the Thing, as well as, the ID_{client} included in the unauthorized request.

[1]Hence, it can simply implemented as a counter



Resource	Policy	SK
R1	URI$_{policy1}$	HMAC$_{MSKA1}$(T1)

Figure 1. System setup

The ACP should authenticate the client, examine if he abides by the access control policy URI_{policy}, and if he is the real owner of ID_{client}. If these conditions hold, the ACP calculates $SK_{acp,ID_{Thing}}$ using equation (1) and then calculates:

$$SK_{client} = HMAC_{SK_{acp,ID_{Thing}}}(URI_{policy}, t, ID_{client}) \quad (3)$$

The outputs of equations (2) and (3) are equal if the following conditions hold:

- The Thing is the legitimate owner of the ID_{Thing}.
- $SK_{acp,Thing}$ of equation (2) is the same as $SK_{acp,ID_{Thing}}$ of equation (3).
- URI_{policy} and token have not been modified when transmitted from the Thing to the client.

Finally, the ACP securely transmits SK_{client} to the client. At the end of this step, the client and the Thing end-up sharing the same secret key. This key can be used for establishing a secure communication channel, e.g., by using a pre-shared key ciphersuite for Transport Layer Security (TLS) [15], with $SK_{session}$, SK_{client}, being the pre-shared key and the token the *identity hint*. During the secure channel establishment, the Thing must use the Token Table and examine whether the token has expired.

When a secure channel has been established, the client sends an *authorized* request for the desired resource. Upon receiving such a request, the Thing must use the Token Table and examine if the resource associated with the secure channel is the same as the one requested by the client. If the latter condition holds, then the Thing responds accordingly. When a transaction is completed, the token and the associated record are removed from the Token Table.

Figure 2 illustrates the unauthorized request and the client authentication phases. In this example, in which it is assumed that the setup phase depicted in Figure 1 has taken place, a client (C1) issues an unauthorized request for the resource R1 provided by T1. Since R1 is protected by $URI_{policy1}$, T1 generates a token (Token1) and calculates the appropriate session key. Then, it updates the Token Table accordingly, by using C1's IP address and port as the session identifier. As a next step, T1 sends $URI_{policy1}$ and Token1 back to C1. C1 establishes a secure communication channel with A1 and sends his identification data along with T1, Token1, $URI_{policy1}$, and C1. Finally, A1 calculates the appropriate secret key and sends it back to C1.

IV. Evaluation

A. Implementation

In this section we present our proof-of-concept implementation. As a Thing we used a Nucleo-f401RE development board. This board uses an ARM Cortex-M4 32-bit CPU with frequency 84MHz, 512KB Flash memory, and 96KB SRAM. We implemented the Thing part of our system using the RIOT operating system [16]. Messages between the Thing and the client were exchanged using the CoAP protocol [17]. CoAP has been designed and developed to be a "lightweight HTTP" so that it can be suitable to operate in constrained IP networks. The CoAP interaction model is similar to the client/server model of HTTP. A CoAP client can issue a request message to a server using a method code on a resource, which is identified by a URI scheme. If the CoAP server is able to serve the request, it responds to the requester with a response code and the payload. The CoAP server, located in the Thing, was implemented using the

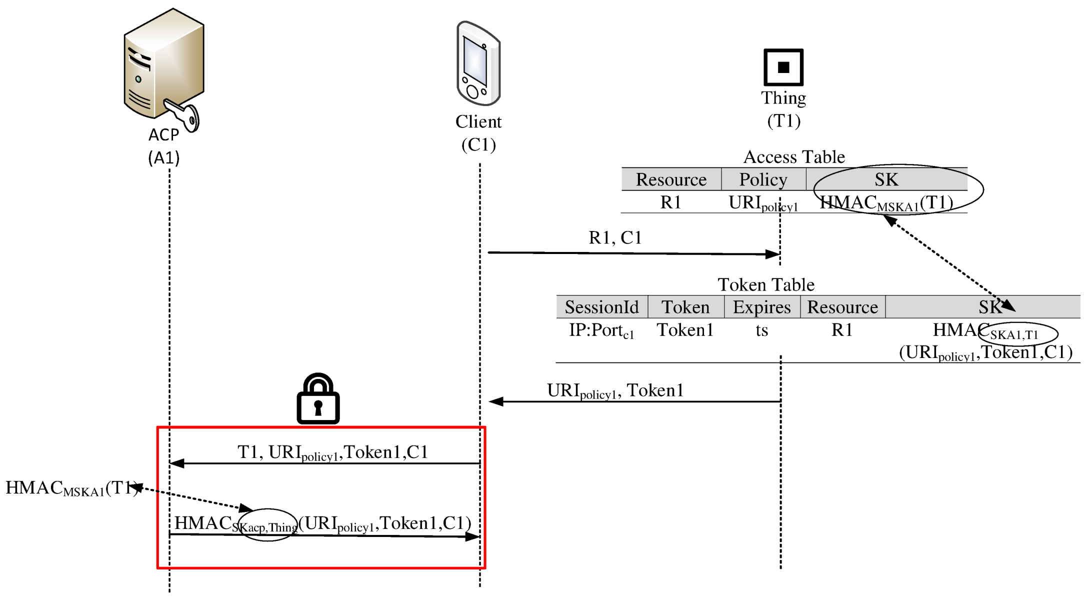

ACP
(A1)
Client
(C1)
Thing
(T1)
Access Table
Resource | Policy | SK
R1 | URI_policy1 | HMAC_MSKA1(T1)
R1, C1
Token Table
SessionId | Token | Expires | Resource | SK
IP:Port_c1 | Token1 | ts | R1 | HMAC_SKA1,T1 (URI_policy1,Token1,C1)
URI_policy1, Token1
T1, URI_policy1,Token1,C1
HMAC_MSKA1(T1)
HMAC_SKacp,Thing(URI_policy1,Token1,C1)

Figure 2. Unauthorized request and client authentication. The secret key of the HMAC used for the calculation of $SK_{session}$ is the value of the $SK_{acp,Thing}$ column of the corresponding row of the Access Table. Similarly the secret key of the HMAC used by A1 to calculate C1's secret key is $HMAC_{MSK_{A1}}$.

microcoap library[2]. The client part of our application was implemented as a J2SE application, using the Californium CoAP framework.[3]

Our implementation considers single resource per Thing. Initially a client sends a CoAP POST request for this resource, including her ID_{client} in the request body. The Thing generates a random token using the random number generator provided by the RIOT OS and sends the appropriate response back to the client. The client communicates with a web server that emulates the ACP using HTTPs and receives back a symmetric encryption key. Finally, and since the RIOT OS does not support DTLS, the client sends a new CoAP post request, including in the POST body the token and the desired resource encrypted with the received key. The Thing decrypts the POST body and if the decrypted token and resource match the corresponding entries stored in the Token Table, the request is accepted. As a $Session_{id}$ we use client's IP address and port.

B. *Performance evaluation*

The computational overhead that our system imposes to Things is related to the computation of an HMAC, as well as, the encryption and the decryption of a message using a symmetric encryption algorithm. For the HMAC calculation we used the SHA-256 based implementation provided by the RIOT OS. As a symmetric encryption algorithm we used AES in CTR mode, also provided by the RIOT OS. We set the $SK_{acp,Thing}$, which is the key used for the HMAC calculations, to be 256bits. Moreover, we used the first 128bits of $SK_{session}$/SK_{client} as the AES key.[4]

Figure 3 shows the time in microseconds required to calculate an HMAC, as well as, the time to perform an AES-CTR encryption/decryption, as a function of the length of the input message (measured in bytes). Our benchmarks were executed 30 times on the same nucleo board. The graphs illustrate the average values (variations were negligible). It can be observed that computation times are negligible.

C. *Security evaluation*

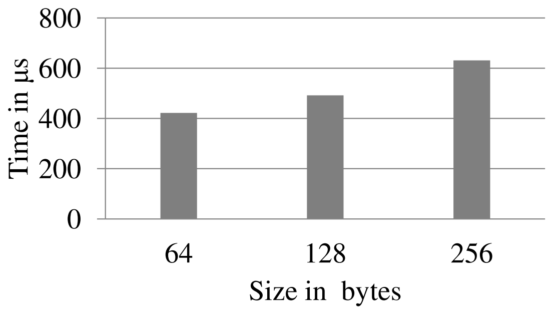
Our system makes the following (security related) assumptions. ACPs store securely clients' credentials, perform proper client authentication, and authorization. Moreover, there exist a secure communication channel between clients and ACPs that protects the confidentiality and the integrity of the exchanged messages, and it provides endpoints identification (when necessary). Clients do not share their ACP credentials, neither the generated symmetric encryption keys. Finally, Things respect ACP decisions and provide resources only to authorized users.

In the following we examine the security of our system under various threat models.

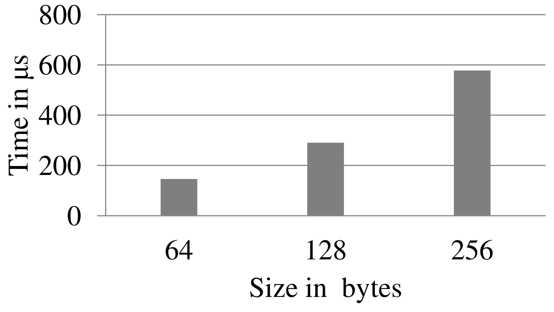
1) Malicious observer: In this threat model a malicious entity simply observes the communication channel between a client and a Thing. The attacker learns the following information: the resource name, the ID_{client}, the URI_{policy}, and the token. It may also be possible for that attacker to learn the ID_{Thing}. Providing that the HMAC used in our

[2]https://github.com/1248/microcoap
[3]https://eclipse.org/californium/

[4]Note, this is only because RIOT does not support DTLS.



(a)



(b)

Figure 3. Time required to perform an HMAC calculation (a), as well as, an AES encryption/decryption (b) as a function of the input length.

system is secure, and that $SK_{acp,Thing}$ is adequate large and random, the attacker is not able to deduce any secret information. Our implementation uses a SHA-256 based HMAC, with key size 256bits.

2) Active attacker: In this threat model a malicious entity observes the communication channel between a client and a Thing, and it is able to manipulate transmitted data. If the attacker modifies the ID_{client}, or the URI_{policy}, or the token, then the outputs of equations (2) and (3) (presented in Section III) will not be the same and as a result the client and the Thing will not end-up sharing the same secret key (therefore, subsequent steps will fail). If the attacker modifies the resource name (included in the client's unauthorized request), then we distinguish the following cases:

- The modified name is not a valid resource name. In that case the client will receive the corresponding error message.
- The modified name is a valid resource name that corresponds to a resource protected by $URI_{policy'}$. If the client is not authorized for $URI_{policy'}$ he will receive an error message from the ACP, otherwise, the client will include a wrong resource name in the authorized request.[5]

[5]It is reminded here that authorized requests are sent over a secure communication channel, therefore the attacker cannot modify the transmitted content.

3) Man in the middle: In this threat model a malicious entity intercepts the communication between a client and a Thing. We distinguish two types of man in the middle attacks. In the first type, the attacker owns secret key generated by the same ACP the protects the targeted resource. This attack is illustrated in Figure 4. In this figure an attacker (M1) intercepts the communication between a client (C1) and a Thing (T1). M1 has a secret key generated by the same ACP as the one that protects the desired resource in C1. Nevertheless, M1 is not a valid Thing identifier for that specific resource. Initially, C1 sends an unauthorized request. M1 forwards to T1 this request either unmodified, or by replacing C1's identifier with its own identifier (i.e., M1). T1 replies with a token (Token1) and a policy URI ($URI_{policy1}$). M1 then, forwards T1's response to C1. Finally, C1 sends an authentication request to the ACP. It should be noted that the Thing identity included in C1's authentication request is T1 and not M1. In the button of Figure 4 there are the keys that each entity may learn/generate (depending on the attacker's choices). In any case, the keys generated by M1, cannot be used to further intercept the communication between C1 and T1.

In the second type of man in the middle attack, the attacker has business relationship with the same ACP as the targeted client. Moreover, this ACP stores the policy that protects the desired resource. This attack is illustrated in Figure 5. In this figure an attacker (M2) intercepts the communication between a client (C1) and a Thing (T1). Initially, C1 sends an unauthorized request. M2 forwards to T1 this request either unmodified, or by replacing C1's identifier with its own identity (i.e., M2). T1 replies with a token (Token1) and a policy URI ($URI_{policy1}$). M2 then, forwards T1's response to C1. At the same time, M2 sends an authentication request to the ACP. Finally, C1 sends an authentication request to the ACP. In the button of Figure 5 there are the keys that each entity may learn/generate (depending on the attacker's choices). In any case, the keys learned by M2, cannot be used to further intercept the communication between C1 and T1. This happens because the attacker cannot include the client's identifier in the authentication message sent in the ACP.

4) Loss of secret keys: Our system considers the following secret keys: the master secret key (MSK) belonging to an ACP, the secret keys that ACPs generate for things ($SK_{acp,Thing}$), and the session keys ($SK_{session}$) which are the keys eventually shared between a Thing and a client after a successful execution of our protocol.

ACP's MSK is used for generating $SK_{acp,Thing}$, therefore the loss of a MSK means that all generated $SK_{acp,Thing}$ keys must be updated. It should be noted however, that clients are not aware of any MSK or $SK_{acp,Thing}$, therefore there is no need for implementing a key revocation procedure in the client side.

$SK_{acp,Thing}$ keys are Thing specific. An entity that

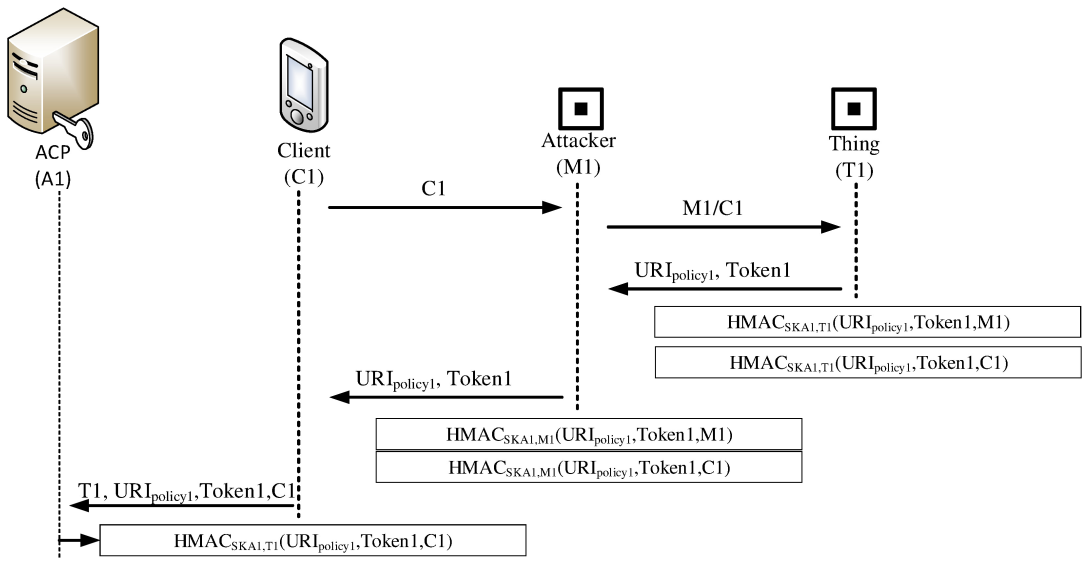

ACP
(A1)
Client
(C1)
Attacker
(M1)
Thing
(T1)
C1
M1/C1
URIpolicy1, Token1
HMACSKA1,T1(URIpolicy1,Token1,M1)
HMACSKA1,T1(URIpolicy1,Token1,C1)
URIpolicy1, Token1
HMACSKA1,M1(URIpolicy1,Token1,M1)
HMACSKA1,M1(URIpolicy1,Token1,C1)
T1, URIpolicy1,Token1,C1
HMACSKA1,T1(URIpolicy1,Token1,C1)

Figure 4. Man in the middle attack. The attacker pretends to be a valid Thing.

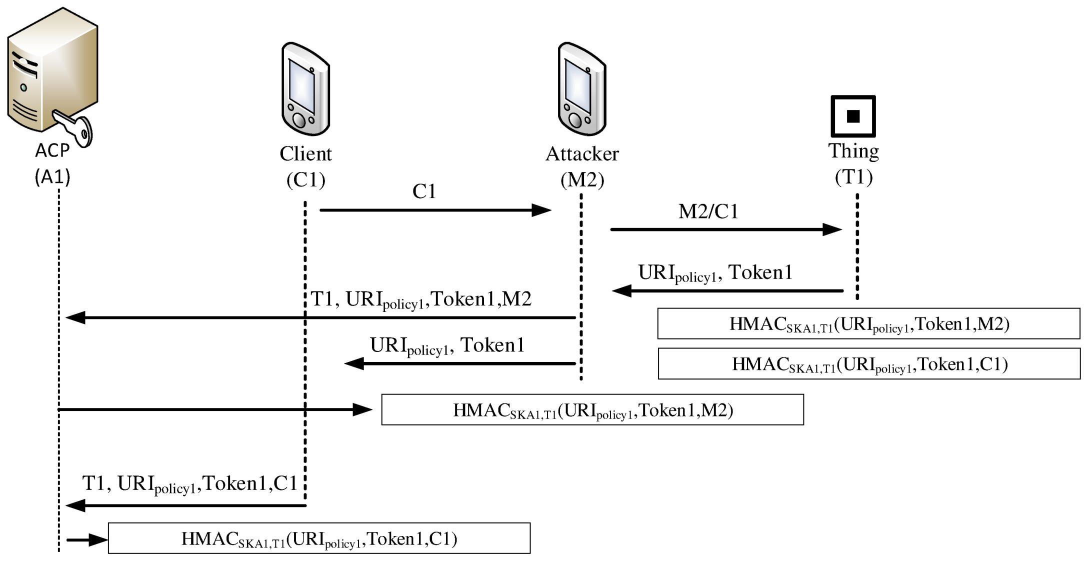

ACP
(A1)
Client
(C1)
Attacker
(M2)
Thing
(T1)
C1
M2/C1
URIpolicy1, Token1
T1, URIpolicy1,Token1,M2
HMACSKA1,T1(URIpolicy1,Token1,M2)
URIpolicy1, Token1
HMACSKA1,T1(URIpolicy1,Token1,C1)
HMACSKA1,T1(URIpolicy1,Token1,M2)
T1, URIpolicy1,Token1,C1
HMACSKA1,T1(URIpolicy1,Token1,C1)

Figure 5. Man in the middle attack. The attacker pretends to be a valid client.

knows this key can intervene, without being detected, the communication between a client and the legitimate key owner. These keys are generated based on the Thing identifier, therefore, in order for this key to be updated, the Thing has to change identifier (usually this means re-flushing the firmware, or replacement of the Thing). If the client software is pre-configured with the Thing identifier, then the software has to be updated. If the client software "discovers" Thing identifiers (e.g., using a service discovery protocol, such as mDNS [18]), then the discovery service should prevent the advertisement of services by Things with revoked identifiers, or clients should maintain a list of revoked Thing identifiers, or ACPs should maintain a list of legitimate Thing identifiers.

$SK_{session}$ keys are ephemeral and they are only used during a particular session. When all transactions of a session are completed these keys are discarded, therefore, compromising such a key does not pose any threat for any subsequent session.

D. Privacy analysis

A feature of the proposed system is that end-user privacy is enhanced. What a Thing learns about a client is that she has business relationships with an ACP, the resource in which she is interested, as well as, a client specific identifier. This information is much less, compared to the information that Things learn when user credentials or access control policies are stored in them. Moreover, in order to avoid client tracking, clients may include a different ID_{client} in every unauthorized request. Of course, this premises that ACPs have a method for mapping all these identifiers to particular client, but this can be easily implemented (e.g., by using hash chains).

Moreover and provided that access control policies are generic, client's interests can be hidden from ACPs. Indeed, an ACP does not have to know in which resource a client is interested in order to authenticate and authorize her. This cannot be achieved, for example, in capabilities-based systems: in these systems the token provider should know what a client wants to access in order to include the appropriate capabilities in the generated capabilities token.

A third party that observes the communication channel between a client and a Thing has access to client's unauthorized requests and the corresponding responses. Since this information is transmitted unsecured it should contain as less information as possible. The initial request should not include any client specific private data (e.g., her location): this information should be included in the authorized request which is transmitted over a secure channel. If the Thing offers a single resource, or many resources protected by the same policy, then the resource name may not be included in the unauthorized request. Similarly, a URI_{policy} should not reveal any information about the client; ideally, the only meaningful information that a 3^{rd} party may learn by observing a URI_{policy} is the network location of an ACP. If a URI_{policy} should contain client specific information, then this information can be encrypted with $SK_{acp,Thing}$ and appended to the URI_{policy}.

V. Discussion

A. Thing identifiers

An essential component of our system is Thing identifiers (i.e., ID_{Thing}). ID_{Thing} are used by ACPs in order to generate the appropriate secret keys (SK). It is of paramount importance for an ACP to be able to tell if the resource owner that requests the generation of a SK for a particular ID_{Thing} is allowed to perform this operation. Failing to verify this binding can result in attackers obtaining the SK of any arbitrary Thing. If an ACP and a resource owner are the same real world entity, then this verification is trivial. This is not an uncommon scenario. For example an enterprise may implement an ACP in order to allow access to its sensors only to its employees. On other hand, general purpose ACPs need a separate mechanism in order to perform this validation. One way to implement such a mechanism is by using as a prefix in ID_{Thing} a resource owner specific identifier, e.g., a domain name, an email address, a public key, a user name in a social network, etc. Then, the ACP has only to validate that the resource owner that requests the SK is the owner of the ID_{Thing} prefix. Depending on the resource owner identifier format there are many and well-studied methods for performing that validation.

Our design implies that each ID_{Thing} is used by a single physical Thing. Nevertheless, nothing prevents a resource owner to use the same ID_{Thing} for many Things. The drawback of this approach is that all these Things will share the same secret key. Therefore, in case of secret key loss, all these Things should be updated (or withdrawn) as described in Section IV-C.

B. The role of token and ID_{client}

Two important variables of our protocol are the token and the ID_{client}.

The role of tokens is to enforce communication with ACPs in every session. If there were not tokens, then all sessions between a Thing and a client would use the same key, therefore, a client could cache this key and use it even if his access rights were revoked.

The role of ID_{client} is to prevent the second type of man in the middle attack. As discussed in section IV-C this type of attack is prevented because an attacker cannot include a client's identifier in an authentication message sent to an ACP. Even though from the Thing's perspective this identifier is a random number, it may not be desirable to use it. An alternative solution for mitigating this type of man in the middle attack could be to have ACPs remembering all tokens that have been generated by a specific Thing and refuse authorization to clients presenting and already used token. So, if this approach was used, in the example of Figure 5, if M2 could be authorized by $URI_{policy1}$ then he could retrieve the secret key from the ACP, hence, the ACP would refuse to generate a secret key for C1. This way C1 would be protected against M2. Of course M2 would be able to access the resource, but this is not a security breach since, in any case and as assumed, M2 is authorized to access it.

C. Multiple ACPs per resource

In our system, each resource is protected by a single policy provided by a single ACP. There are cases where it is desirable to have a resource protected by multiple policies, each provided by a different ACP. For example, the case of a Thing that is an actuator that opens and closes the door of a shared parking space used by the personnel of two companies (namely *A* and *B*). In this case, it is desirable for each company to maintain its own ACP, in order to protect their personnel information. Our system can be adapted for this scenario as follows.

For each resource the Access Table is modified to contain pairs of $[URI_{policy}, SK]$. For example in the above scenario an Access Table could be of the form:

Resource	**Policy**	**SK**
ControlDoor	$A_{ACP}/Policy1$	$SK_{A,thing}$
	$B_{ACP}/Policy8$	$SK_{B,thing}$

When a client issues an authorized request, the Thing responds with the token and a *ordered* list of the available URI_{policy}. Moreover, the Thing does not generate the session SK at this stage (hence it leaves the corresponding field of the Token Table empty). When the client obtains the appropriate secret key, she includes in her authorized request, in plaintext, which element of the ordered list of the available URI_{policy} she has selected. For example, if DTLS is used, this can be done by prefixing a number in the token (remember here that the token can be used as the identity hint). At this point, the Thing constructs the corresponding session SK and resumes the execution of our protocol without any further modification.

D. Business opportunities

We believe that the flexibility and the extensibility of our system create the potential for new business opportunities.

ACPs can be easily built on top of existing user management systems. Moreover, once an ACP is built, it can be used for many applications and diverse systems (not necessarily IoT specific). For example, the same ACP (and even the same URI_{policy}) can be used for controlling the parking system of a company, entrance to offices, access to coffee machines, or even access to files. Of course, we should not ignore that ACPs can also be provided by 3rd parties. Enterprises such as social networks, security companies, and others may expand their business by providing this new additional service even if they have not invested in the IoT.

Our system facilitates cross business services. Deployed functions can be easily offered to customers of other enterprises. Co-operating enterprises do not have to reveal any sensitive information to each other, since the only information they provide is URIs to access control policies. Furthermore, access control policies can be easily managed: access control policies can be modified by an ACP without any communication with the Things in which protected resources are stored, this way end-users can be easily added or removed from a system. Finally, an administrator can easily establish or terminate a business relationship by adding or removing a URI_{policy} from an Access Table.

Application development is also facilitated by our system. Firstly, access control is provided as a new layer on top of the provided resources, therefore, resources development does not need to have access control in mind. Secondly, the Things-part of our system is lightweight and straightforward to implement, therefore, Thing developers can easily incorporate our solution in their systems. Thirdly, ACPs can implement and provide access control policies without having to be aware of the resources that use their policies.

VI. Conclusions and future work

In this paper, we proposed a solution to the thorny problem of access control in the IoT. Our solution enables delegation of access control decisions to trusted third parties, therefore it unburdens Things from storing end-user identification data and from evaluating access control policies. The proposed system is lightweight and the only computational intensive operations that a Thing should perform are HMAC computations and symmetric key encryptions; our evaluation on a real world development board shows that this overhead is minimal. Compared to existing systems, our solution reveals less client-related information to the Things and to the ACPs, hence it enhances end-user privacy. Moreover, in our system, policy modification involves no communication with the Things and the clients. By using our system a Thing and an authorized client can agree on a shared secret key, without any other pre-configured secret. Moreover, the secret key generation process allows clients to verify a Thing's identifier, without needing public key encryption. Finally, our system can be easily incorporated into existing applications since it clearly separates application logic from access control decisions, creates new business opportunities (e.g., for ACPs), and facilitates cooperation among various stakeholders.

In addition, since our system implementation was limited by the lack of DTLS support by the RIOT operating system, we were led to develop our own communication protocol, on top of CoAP, in order to implement authorized resource requests. However, if DTLS were supported, the commonly agreed secret key could be used for establishing a secure communication channel.

Finally, our system assumes that resource owners and clients co-operate with the same ACPs. It is in our future plans to examine "interdomain" access control policies scenarios, i.e., scenarios where a resource is protected by an access control policy stored in an ACP_A, a client has business relationships with an ACP_B, ACP_A and ACP_B are bilateral trusted, hence, the client can access the resource.

References

[1] N. Fotiou, G. F. Marias, and G. C. Polyzos, "Access control enforcement delegation for information-centric networking architectures," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 497–502, Sep. 2012.

[2] N. Fotiou, A. Machas, G. C. Polyzos, and G. Xylomenos, "Access control delegation for the cloud," in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on.* IEEE, April 2014, pp. 13–18.

[3] Z. Shelby, K. Hartke, and C. Bormann, "Delegated CoAP Authentication and Authorization Framework (DCAF)," IETF, Internet-Draft, 2015.

[4] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[5] E. Yuan and J. Tong, "Attributed based access control (ABAC) for Web services," in *IEEE International Conference on Web Services (ICWS'05)*, July 2005, pp. 561–569.

[6] C.-T. Lee, C.-H. Yang, C.-M. Chang, C.-Y. Kao, H.-M. Tseng, H. Hsu, and P. H. Chou, "A smart energy system with distributed access control," in *Proceedings of the IEEE International Conference on Internet of Things, Taipei, Taiwan*, 2014.

[7] D. Hardt (ed.), "The OAuth 2.0 authorization framework," IETF, RFC 6749, 2012.

[8] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari, "Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios," *IEEE Sensors Journal*, vol. 15, no. 2, pp. 1224–1234, Feb 2015.

[9] D. Recordon and D. Reed, "OpenID 2.0: a platform for user-centric identity management," in *Proceedings of the second ACM workshop on Digital Identity Management*, ser. DIM '06, New York, NY, USA, 2006, pp. 11–16.

[10] A. Blazquez, V. Tsiatsis, and K. Vandikas, "Performance evaluation of openid connect for an iot information marketplace," in *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, May 2015, pp. 1–6.

[11] B. Anggorojati, P. Mahalle, N. Prasad, and R. Prasad, "Capability-based access control delegation model on the federated iot network," in *Wireless Personal Multimedia Communications (WPMC), 2012 15th International Symposium on*, Sept 2012, pp. 604–608.

[12] S. Gusmeroli, S. Piccione, and D. Rotondi, "A capability-based security approach to manage access control in the internet of things," *Mathematical and Computer Modelling*, vol. 58, no. 5, pp. 1189 – 1205, 2013, the Measurement of Undesirable Outputs: Models Development and Empirical Analyses and Advances in mobile, ubiquitous and cognitive computing.

[13] L. Seitz, G. Selander, and C. Gehrmann, "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*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 1–6.

[14] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, Dec. 1978.

[15] P. Eronen and H. Tschofenig, "Pre-shared key ciphersuites for transport layer security (TLS)," IETF, RFC 4729, 2005.

[16] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, April 2013, pp. 79–80.

[17] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, 2014.

[18] S. Cheshire and M. Krochmal, "Multicast DNS," 2013, RFC 6762, 2005.