A Lazy Approach to Access Control as a Service (ACaaS) for IoT

An AWS Case Study

Tahir Ahmad
Security & Trust Unit, FBK-ICT, Trento, Italy
DIBRIS, University of Genova, Italy
ahmad@fbk.eu

Umberto Morelli
Security & Trust Unit, FBK-ICT, Trento, Italy
umorelli@fbk.eu

Silvio Ranise
Security & Trust Unit, FBK-ICT, Trento, Italy
ranise@fbk.eu

Nicola Zannone
Eindhoven University of Technology, The Netherlands
n.zannone@tue.nl

ABSTRACT

The Internet of Things (IoT) is receiving considerable attention from both industry and academia because of the new business models that it enables and the new security and privacy challenges that it generates. Major Cloud Service Providers (CSPs) have proposed platforms to support IoT by combining cloud and edge computing. However, the security mechanisms available in the cloud have been extended to IoT with some shortcomings with respect to the management and enforcement of access control policies. Access Control as a Service (ACaaS) is emerging as a solution to overcome these difficulties. The paper proposes a lazy approach to ACaaS that allows the specification and management of policies independently of the CSP while leveraging its enforcement mechanisms. We demonstrate the approach by investigating (also experimentally) alternative deployments in the IoT platform offered by Amazon Web Services on a realistic smart lock solution.

CCS CONCEPTS

• **Security and privacy → Access control**; **Authorization**;

KEYWORDS

Internet of Things; Policy specification and management; Attribute-Based Access Control; IoT platforms; Edge Computing

ACM Reference Format:
Tahir Ahmad, Umberto Morelli, Silvio Ranise, and Nicola Zannone. 2018. A Lazy Approach to Access Control as a Service (ACaaS) for IoT: An AWS Case Study. In *SACMAT '18: The 23rd ACM Symposium on Access Control Models & Technologies (SACMAT), June 13–15, 2018, Indianapolis, IN, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3205977.3205989

1 INTRODUCTION

The Internet of Things (IoT) holds the promise to bring huge benefits for users, industry, and society by combining the capabilities of collecting large amount of data over long periods of time with substantial processing capabilities and low-latency communication. However, the convergence of these technologies raises significant security and privacy concerns. To mitigate these risks, access control plays a key role for providing controlled information sharing—a necessary condition to build privacy into IoT solutions—and integrity guarantees—a crucial pre-requisite for the correct functioning of an entire IoT system.

Traditional approaches to access control (see, e.g., [16]) are not adequate for IoT due to several sources of complexity, including the heterogeneity and large number of connected devices (e.g, different types of sensors distributed in many locations), resource constraints (on processing, storage and communication), interaction patterns (from stable and long-lived to casual and short-lived), and augmented context awareness (such as time, location, and mode of operation of a system) [13]. As a result, the management and enforcement of access control policies become daunting tasks that hinder the deployment of secure and privacy-aware IoT solutions with negative consequences on their adoption by users because of a lack of trust.

In an attempt to combine large scale data processing and low-latency communication for IoT solutions, Cloud Service Providers (CSPs)—such as Amazon, Google, and Microsoft—started extending their cloud computing platform with support for edge computing. At the same time, CSPs offer refinements of their access control mechanisms with the aim of satisfying the requirements posed by IoT solutions on the management and enforcement of security policies. Unfortunately, such offerings are not satisfactory as they suffer serious drawbacks such as: *(D1)* limited support for policy administration, *(D2)* proprietary policy languages (which increase the risk of vendor lock-in), and *(D3)* limited expressiveness in specifying complex authorization conditions that depend on a multitude of resources and contextual attributes.

To alleviate drawbacks (*D1*), (*D2*) and (*D3*), we propose a *lazy* approach to Access Control as a Service (ACaaS) tailored to IoT solutions and built on top of existing IoT platforms. ACaaS allows one to outsource the administration and enforcement of access control policies to a trusted third party. The advantages of ACaaS are several and include a comprehensive and uniform support for policy administration—addressing (*D1*) together with an expressive and high-level (independent of a particular CSP) policy specification language—addressing both (*D2*) and (*D3*).

The main difference between ours and most existing ACaaS solutions [1, 6, 7, 10] is that we outsource the management but not the

SACMAT '18, June 13–15, 2018, Indianapolis, IN, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5666-4/18/06…$15.00
https://doi.org/10.1145/3205977.3205989

enforcement of policies and prefer to reuse the enforcement mechanisms provided by each CSP. Following and extending the approach introduced in [12], we do this by translating a high-level policy language based on Attribute Based Access Control (ABAC) [9] to the policy language adopted by a given CSP. This allows us, on the one hand, to reuse well-engineered, robust, and tested enforcement mechanisms and, one the other hand, reduce the overhead due to the invocation of an external evaluation point for every authorization request. An additional advantage is in speeding up authorization request evaluation by exploiting the support for edge computing that is available in most IoT platforms from CSPs.

Two remarks are in order. First, we have chosen ABAC as the underlying access control model for the high-level language used in our ACaaS solution as recommended by previous works (see, e.g., [3]) that argue for its adequacy to express context-dependent authorization requirements of IoT systems. Second, we have chosen the Amazon Web Service (AWS) IoT platform to conduct our experiments since it is one of the more advanced solution available on the market and allows us to perform a thorough validation of the flexibility and adequacy of our approach.

IoT solutions can be developed in many and heterogeneous application domains. Each domain induces a different emphasis on the security requirements to be met (see, e.g., [13] for an in-depth discussion). For instance, in healthcare and smart home applications, confidentiality, integrity, and privacy are crucial as the data collected by smart devices are personal and sensitive. Instead, for smart community services, availability becomes of paramount importance whereas confidentiality and privacy are less relevant as the data collected by the devices are usually less sensitive. Since access control has been traditionally used to guarantee confidentiality and integrity, we have selected a smart lock system in the smart home domain as the reference application to elicit the requirements of the access control solutions for IoT. We also discuss scalability and availability issues when the smart lock system is lifted to the enterprise level in the context of hotel chains or geographically distributed company buildings.

The remainder of the paper is organized as follows. Sec. 2 introduces the smart lock system and the requirements for access control. Sec. 3 briefly discusses the AWS IoT platform and its shortcomings. Sec. 4 presents our lazy approach to ACaaS, illustrates four possible deployments, and shows how it supports the smart lock system. Sec. 5 evaluates the approach both qualitatively and experimentally with respect to the requirements identified in Sec. 2. Sec. 6 discusses related work, and Sec. 7 draws some conclusions and summarizes future work.

2 USE CASE & REQUIREMENTS

We describe a smart lock system as a realistic use case scenario, identify the requirements that access control solutions for IoT should meet, and guide future developments of similar solutions.

2.1 Smart lock system

Smart locks are cyber-physical devices that aim to replace traditional locks with smart cylinder remotely controlled through a mobile application or a web portal. They can be deployed by individuals (in homes) or enterprise customers (typically hotels) to reduce management costs and improve service quality; they can also be used as a smart work enabler (in smart offices) or to provide innovative services. Amazon Key in-home delivery service[1] is one such innovative service that is based on cloud-based Amazon camera and a smart door lock. Using this service, the owner can authorize a delivery company to temporary gain access to his home and monitor the activity using a cloud-based camera. The service can be expanded to include other in-home services such as house cleaning, pet sitters, etc. Similarly, Sofia locks[2] provide smart lock solutions for residential, commercial, industrial and public buildings.

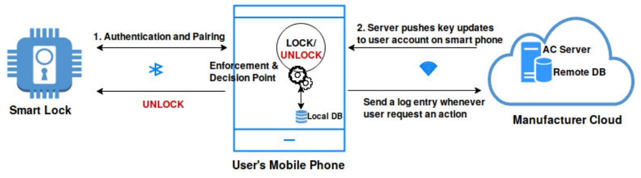
1. Authentication and Pairing
UNLOCK
Smart Lock
LOCK/
UNLOCK
Enforcement & Decision Point
Local DB
User's Mobile Phone
2. Server pushes key updates to user account on smart phone
Send a log entry whenever user request an action
AC Server
Remote DB
Manufacturer Cloud

Figure 1: Architectural design of the smart lock system

Our use case is inspired to real world smart lock solutions that we had the opportunity to analyze and it features their most important characteristics. The architectural design of most commercial smart lock systems is based on a centralized IoT architecture in which the application logic is governed by a central entity (deployed in a private cloud) that provides a limited set of well-known entry points (e.g., APIs). Figure 1 presents an abstract view of the architecture. The main components are an electronically augmented deadbolt, which includes a smart cylinder lock and a controller, and a user mobile phone acting as an Internet gateway. The smart lock lacks direct Internet connectivity and, thus, relies on the user's mobile phone to communicate with the manufacturer cloud when the phone enters the Bluetooth range of the lock.

The smart lock employs an access management system, deployed in the manufacturer's private cloud, where smart lock owners can configure user permissions through an API. The access management system is based on Group Based Access Control (GBAC) [5] in which the access of users to resources is regulated using the notion of group, i.e. a logical collection of one or more entities that have common properties. The owner can add/remove a person to one of the groups predefined by the smart lock manufacturer (e.g., "Owner", "Resident", "Recurring Guest", "Temporary Guest"). Listing 1 shows a simple access rule of the smart lock system. This access rule indicates that users belonging to `persongroup` "Temporary Guest" can open doors in `doorgroup` "V3I6G5LSQGWL". Field `accessprofile` indicates the time scheme in which the rule is applicable (e.g., during office hours). A default `accessprofile` "*always*" is assigned if this field is not defined. Fields valid_from and valid_to denote the validity of the permission. The list of authorized users is also maintained in the local database of the smart lock (see below for the process used to update the smart lock database).

The smart lock system also provides an access logging mechanism. Whenever a user interacts with the lock, it sends a log entry recording the action, the user who performed it and the timestamp to the logging mechanism hosted in the manufacturer cloud. Sample access logs are shown in Listing 2, where `token_uuid` identifies the

[1] https://www.amazon.com/key
[2] https://www.sofialocks.com/it/smartlocks/

Listing 1: Access Control Policy

```
<xml version="1.0" encoding="UTF-8">
<accessrule>
  <description>description of customer</description>
  <doorgroup >V3I6G5LSQGWL</doorgroup>
  <accessprofile>18HLFPN293</accessprofile>
  <persongroup>Temporary Guest</persongroup>
  <valid_from>2018-01-06 00:00:00+00:00</valid_from>
  <valid_to>2018-12-06 23:59:00+00:00</valid_to>
</accessrule>
```

Listing 2: Access Logs

```
<xml version="1.0" encoding="UTF-8">
<accesslog>
  <token_uuid>2WSROEJSJ72R</token_uuid>
  <code>1</code>
  <timestamp>2018-01-03 12:27:03</timestamp>
  <lockid>30EBG7RQG12R</lockid>
</accesslog>
<accesslog>
  <token_uuid>2WSROEJSJ72RP</token_uuid>
  <code>2</code>
  <timestamp>2018-01-03 12:28:17</timestamp>
  <lockid>30EBG7RQG12R</lockid>
</accesslog>
```

user and `code` refers to the action performed. In the listing, `code` 1 corresponds to UNLOCK and `code` 2 to LOCK. Field `timestamp` records the date-time when the action was performed, and field `lockid` indicates the specific lock on which the action was performed.

Below we describe the key processes supported by the smart lock system.

- **User Registration & Permission Configuration:** The smart lock manufacturer provides users with a mobile application and a unique authorization code along with the smart lock. After installing the application, the owner pairs the application with the smart lock using the provided unique authorization code. Then, the owner can generate short term and long term digital keys for various types of users (family members, visiting friends, etc.) by accessing the access management system in manufacturer's private cloud. The access to the private cloud is controlled with the usage of One Time Password (OTP) generated by the application on the owner's mobile device.
- **Locking and Unlocking Process:** The smart lock is controlled through a smart lock application provided by the manufacturer and installed on the user mobile device. A user can LOCK or UNLOCK the lock through the smart lock's mobile application. As shown in Figure 1, when a user enters the Bluetooth range of the smart lock, his mobile phone is authenticated and paired with the smart lock through the Bluetooth protocol. Once connected, the smart lock receives the key status of the specific user from the remote access management system via the user's mobile device and uses the received key status to determine whether access should be granted. The key status is also stored in the local database of the smart lock. In case the remote access management system cannot be reached, the smart lock system ensures availability and maintain access by making decision based on the entries in its local database.
- **Key Revocation Process:** The owner is allowed to detach his or any other (authorized) device from the smart lock by accessing the access management system. Specifically, the owner can revoke access from a user by revoking the key assigned to that user and updating the key status inside the key repository in the access management system on the manufacturer cloud. The changes are then propagated to the local database of the smart lock system when the user connect to the lock through his mobile device.

2.2 Analysis of the Use Case Scenario

We now analyze the smart lock system and discuss its limitations. First, we focus on the security issues affecting the adopted access control mechanism and, then, we consider other aspects that can influence the design of an access control solution for IoT.

The access management system provided within the smart lock system only allows smart lock's owners to specify simple policies in the GBAC model. Specifically, owners can assign users to predefined groups and define their permissions with respect to group(s) they belong to. Although this model provides users with a simple and intuitive approach for policy specification, it is rather limited in the policies that can be specified and it is not suitable when fine grained control is needed or access should be granted decision under certain contextual conditions. For example, it is not possible to specify that access should be granted to a temporary guest only if a member of the "Resident" group is at home.

Moreover, the smart lock system has intrinsic vulnerabilities and weaknesses in its design that can be exploited by users to compromise the system:

V1: The smart lock lacks direct connectivity to the Internet and relies on the user's smart phone to interact with the manifacturer's cloud. Therefore, the smart lock implicitly trusts the user to behave faithfully.

V2: The smart lock receives key status updates from the remote access management system only when a user interacts with the smart lock. Moreover, the received updates only concerns the interacting user. Therefore, the smart lock remains unaware of changes in the policies while it cannot connect to the manufacturer's cloud (via the user's smart phone).

V3: The access control logic is implemented in the access management system hosted in the manufacturer's cloud whereas access control policies are enforced locally by the smart lock. If the smart lock is unable to retrieve key status updates, it uses the policies stored in the local database, which however can be outdated (V2).

V4: The smart lock owner can grant, update or revoke a digital key through the remote access management system. However, these actions are subject to a final approval through the mobile application on the owner's mobile phone.

We hereby present two threat models that are typical for smart lock systems like the one described in our scenario.

(1) *Control of a user's mobile device*: The adversary is assumed to be a legitimate user of the system or to be in control of the mobile device of a legitimate user. Therefore, the adversary can block the connectivity between the smart lock and the manufacturer's cloud, for instance, by turning ON airplane mode when interacting with the smart lock.
(2) *Control of the owner's mobile device*: The adversary is assumed to be in control of the owner's mobile device. Besides the capabilities described in the previous threat model, the adversary is

also in control of the application to configure the smart lock system installed on the owner's mobile device.

Next, we discuss some attacks, also identified by [8], based on the aforementioned threat models and the vulnerabilities of the access control mechanism adopted within the smart lock system.

Revocation Evasion: An adversary can exploit the above mentioned vulnerabilities to retain access to the smart lock when his permissions have been revoked by blocking the connectivity between the smart lock and the access management system. Consider, for instance, a housekeeper that has recently been relieved from duty. Accordingly, the owner revokes her permissions for entering his house by performing the key revocation procedure described above. However, by exploiting vulnerabilities V1, V2 and V3 of the smart lock system, the housekeeper can still maintain (unauthorized) access. In particular, she can turn ON the airplane mode on his mobile phone when interacting with the smart lock, thus preventing the smart lock from receiving key status updates. Due to vulnerabilities V1 and V2, the smart lock remains unaware of the revoked permissions. Since the smart lock makes decisions based on the policies in the local database when it is unable to contact the access management system (V3), the housekeeper is still able to enter the apartment.

Logging Evasion: An adversary, who can block the connectivity between the smart lock and the manufacturer's cloud, can also hide his interaction with the smart lock by blocking log messages from reaching the access management system by simply turning ON the airplane mode when interacting with smart lock.

Update Evasion: In case the smart lock's owner looses his mobile device or his mobile device is stolen, he has to remove the device from the smart lock system. To this end, he can access the access management system and revoke the digital key assigned to that device. However, if the adversary posses the owner's mobile device, he can ignore the request for approving the revocation (V4) and, thus, he can access the lock system using the owner's mobile device. In these situations, a user often has no other option than returning the smart lock back to the manufacturer, as happened in the case of Lockstate after sending users a wrong firmware update [11].

Besides the security considerations above, other orthogonal aspects should be considered when designing an access control solution for IoT.

Management: Smart lock solutions not only can be deployed on small scale in homes and small offices, as described in our scenario, but also on larger scale, for instance, in industrial setups and hotels. This requires an access control mechanism to be able to manage the access for a potentially large number of smart locks. The analyzed smart lock system allows assigning a smart lock to a single user account. This means, for instance, that a hotel manager has to manage a large number of accounts, one for each smart locks. This solution is clearly impractical when deployed on large scale. Moreover, policy specification is known to be difficult and error-prone [18]. For instance, when a policy is updated, it is difficult to determine whether the revised policy works as intended. Even small errors can lead to unauthorized accesses. Ensuring the correctness of access control policies is thus a crucial task to guarantee the security of the smart lock system. Finally, the smart lock system provides very little support in the configuration of security mechanisms. For instance, it does not allow the smart lock's owner to configure the access logging system. This can affect user privacy as he cannot prevent his or any other user's interaction with the smart lock to be recorded (recall that access logs are stored in the manufacturer's cloud and, thus, he has not control over them).

Latency: Users standing in front of a door typically expect a response from the smart lock in the order of milliseconds. In our system, the smart lock has to retrieve the key status from the access management system hosted on the manufacturer's cloud to determine whether a user is allowed to open the door. However, the response time from a cloud might vary from milliseconds to seconds or even minutes depending on the geographic location of the cloud. This can affect the functioning of the system as well as user satisfaction in the smart lock solution. Therefore, it is important to consider the response time required by the specific IoT application [4] and to guarantee that the access control solution does not introduce an intolerable delay for users.

Platform-Independence: The analyzed smart lock system is bounded to the manufacturer's private cloud. This, together with the employment of ad-hoc mechanisms, makes the portability of smart lock configurations and policies to another cloud service provider difficult, if possible at all. This is known as *vendor lock-in* and is one of the main issues to the widespread adoption of cloud-based services and applications [6].

2.3 Requirements for Access Control

Based on the discussion above, we have identified a number of requirements to guide the development of access control solutions for smart locks and similar IoT applications (cf. Table 1).

(AC1) **Expressibility:** An access control system for IoT should be applicable in all security contexts by allowing the specification of policies that fit the desired level of granularity. In fact, many IoT applications require enforcing access restrictions that depend on several attributes of users, resources, and the environment. It is thus desirable from an access control policy to be expressive enough to capture the access restrictions to be enforced. In the case of our smart lock system, for instance, the smart lock owner should be able to specify that access should be denied to temporary guests if no member of "Resident" group is at home.

(AC2) **Administration:** The way in which an access control system is configured and managed is very critical to ensure security and privacy within IoT systems. The definition of access control policies is far from being a trivial process due to the interpretation of complex and ambiguous security polices that have to be translated into well-defined, unambiguous and enforceable rules. This requires the access control system to provide users with an administration point for easy translation of security requirements into enforceable access control policies and for the verification of their correctness. The administration point should also provide users with capabilities for the configuration of security mechanisms.

(AC3) **Portability:** Besides affecting the administration of access control policies, the inherent difficulties in defining access control restrictions have also an impact on the migration of the smart lock system across different cloud service providers (CSP), resulting in vendor lock-in. Consider, for instance, a chain of hotels with branches in different parts of the world, where branches in different

Table 1: Requirements of Access Control Systems for IoT

ID	Requirement	Description
AC1	Expressibility	The access control system must allow users to specify fine-grained access control policies.
AC2	Administration	The access control system must provide an administration point to easily configure policies for connected devices and available resources.
AC3	Portability	The access control system needs to be platform independent.
AC4	Extensibility	The access control system must support the enforcement of arbitrary security constraints.
AC5	Latency	The access control system must be designed according to the latency requirements of the IoT application.
AC6	Reliability	The access control system must provide a reliable access decision in every system state.
AC7	Scalability	The access control system must be able to handle a growing number of devices and amount of data generated and processed by those devices.

countries rely on a different CSP, e.g. for legal and/or economic reasons. Each CSP can adopt a different access control mechanism along with a different policy language, which makes the management of smart locks across different branches difficult as policies have to be specified with respect to each CSP. This raises the need of portability for access control policies so that a user can specify policies that can be reused across different CSPs.

(AC4) **Extensibility:** To maximize its viability, an access control system should provide extensibility points to customize policy evaluation with respect to the needs of the application domain. The most noteworthy extensibility point is the possibility to augment the access control system with event driven functions for the evaluation of custom constraints in access control policies [10].

(AC5) **Latency:** Service provision in a cloud-centric IoT architecture might lead to congestion and arbitrary delays due to the large number of requesting services and devices that generate and consume data [15]. Several IoT applications have stringent latency requirements, which impose constraints also on data transfer and decision making processes. To address these concerns, new trends are emerging to move part of the computational logic closer to the physical devices. In particular, new computing paradigms like edge computing and fog computing propose to move cloud capabilities towards network edge to minimize the need to interact with the cloud [19]. However, there is a gray scale between the two extremes – pure cloud and pure edge – that allows a spectrum of possible architectures to distribute the access control logic and responsibilities. The choice of the type of architecture should be driven by the requirements of the IoT application at hand.

(AC6) **Reliability:** The use of cloud has also an impact on the security of the system and, in particular, on the reliability of access decisions. As shown in our scenario, the lack of connectivity between the smart lock and the access management system hosted in the manufacturer's cloud can be exploited by an adversary to maintain the access to the smart lock when his permissions have been revoked (revocation evasion). We advocate that an access control mechanism should be reliable in every system state.

(AC7) **Scalability:** The exponential growth of IoT deployments is highly expected in terms of new devices and amount of data generated and processed by these devices. In our scenario, each smart lock is managed by a single account. However, in case of deployment on large scale (e.g., hotels or industrial setups), the management of smart locks might become a serious concern. Therefore, we envision that an access control mechanism for IoT should be able to scale in size, structure and number of users and resources.

3 AWS IoT

First, we give a brief overview of the Amazon Web Services platform for IoT (AWS IoT). Then, we discuss the problems we encountered in realizing the smart lock system of Sec. 2 on top of AWS IoT.

3.1 AWS IoT & Greengrass

AWS IoT is an extension of the Amazon Web Services platform for IoT. This platform provides support to collect and analyze data from Internet-connected devices and to connect those data to AWS cloud applications, allowing to tie data into applications. The dashed (rounded) rectangle labeled with "DM1: Pure Cloud Architecture" in Fig. 2 shows a high-level view of the AWS IoT architecture with the following components:

- **Device Gateway:** enables IoT devices to securely and efficiently communicate with AWS IoT.
- **Message Broker:** provides IoT devices with a secure mechanism to publish and receive messages to and from each other using the MQTT protocol.
- **Lambda Function:** is a stateless piece of code whose execution can be triggered by a wide range of sources, both internal and external to AWS like web and mobile applications.
- **AWS Management Console:** is a web application providing an built-in user interface for the management of AWS services.
- **Authentication:** AWS IoT provides mutual authentication and encryption to every connected IoT device. It supports two authentication methods: X.509 certificates and Custom Authorizers (authentication based on custom tokens).
- **Access Control:** AWS IoT employs an access control mechanism based on a limited variant of ABAC. An AWS policy is a JSON file that is attached to the certificate of an entity and comprises three main parts: *Effect* (allow or deny), *Action* (e.g., IoT:publish) and *Resources* (e.g., an AWS resource name). A policy can also include a *Condition* that refines the scope of a permission and may contain up to three attributes of the entity. Listing 3 shows a sample AWS access control policy for the smart lock scenario. This policy allows operations "Connect" and "Subscribe" on any resource ("*") whereas it allows operations "Publish" and "Receive" only on the device with *ID* "Thing_ID_1" provided that it belongs to Alice (*Owner*) and is associated to "Room_1" (*Room*). For more details on AWS IoT access control policies, see [3].

To overcome potential latency issues (that are typical of pure cloud platforms for IoT), AWS IoT provides an additional service, called AWS Greengrass, that integrates edge computing. IoT devices connected to the same Greengrass instance (typically on the same network) can be configured as a Greengrass group. In case an IoT device loses connection with the back-end cloud, it can continue to communicate with other IoT devices in the same Greengrass

Listing 3: AWS IoT policy for the smart lock use case

```
{"Version": "2012-10-17",
 "Statement": [
 {"Effect": "Allow",
  "Action": ["iot:Connect","iot:Subscribe"],
  "Resource": "*" },
 {"Effect": "Allow",
  "Action": ["iot:Publish","iot:Receive"],
  "Resource":"arn:aws:iot:us-west-2:X:topic/Test",
  "Condition": {
    "StringEquals": {
      "iot:Connection.Thing.Attributes[Owner]":"Alice",
      "iot:Connection.Thing.Attributes[ID]":"Thing_ID_1",
      "iot:Connection.Thing.Attributes[Room]":"Room_1"}}}
         ]}
```

group over the local network. Greengrass maintains a subscription table defining the messages that can be exchanged within a Greengrass group, where each entry in the subscription table specifies a source (message sender), a target (message recipient) and a topic over which messages can be sent/received. Messages can be exchanged only if an entry exists in the subscription table matching the source, target, and topic; this provides a rudimentary access control mechanism.

3.2 Limitations of AWS IoT & Greengrass

We discuss the limitations encountered in using AWS IoT and Greengrass to realize the smart lock system in Sec. 2 with respect to the requirements in Tab. 1:

- AWS IoT does not allow the specification of fine-grained access control policies (AC1). AWS IoT imposes a restriction on the number of attributes that can be used for policy specification. Only three attributes of the entity can be used in a policy, resulting in coarse grained access control policies, whereas the dynamic nature of IoT might demand to express more complex authorization conditions involving a larger number of attributes and/or refer to properties of the requested resource and environment as in the policy of Sec. 2.2.
- AWS IoT provides limited support for policy administration (AC2). Every service has a different administrative interface inside the AWS Management Console, making policy administration across services cumbersome. Moreover, the policy specification interface provides very little assistance in policy verification. In particular, it does not provide any mechanism to validate and debug the specified policy before enforcement; it only warns the user in case of a syntactic problem.
- AWS IoT uses a proprietary access management system that employs an ad-hoc language for policy specification (cf. Listing 3), thus hindering the migration to other IoT platforms and resulting in vendor lock-in (AC3).
- The use of AWS Greengrass can potentially help meet latency (AC5) and reliability (AC6) requirements by bringing the access logic closer to physical devices. However, Greengrass relies on a rudimentary access control mechanisms based on subscriptions, which does not allow for fine-grained control (AC1). Moreover, it limits the number of IoT devices that can be configured within a deployable instance to 200 and restricts to 50 the number of those who can receive messages from AWS IoT.
- AWS IoT and Greengrass access control mechanisms can be extended by using Lambda functions; unfortunately, the burden of doing this is entirely left on the shoulder of programmers with little or no assistance (AC4).
- To the best of our knowledge, AWS IoT has only been tested with small scale deployments whereas large scale deployments (AC7) with different set of requirements as the ones given in Section 2.3, are still unclear [17].

4 A LAZY APPROACH TO ACaaS

We now explain how to overcome the limitations of the AWS IoT platform discussed in Sec. 3.2 using our lazy approach to ACaaS.

First of all, we observe that the first four requirements in Tab. 1 are readily satisfied by adopting ACaaS. In fact, by using standard policy specification languages (such as XACML) usually based on the Attribute Based Access Control (ABAC) model [9], existing ACaaS solutions support the expressiveness necessary to specify fine grained access control policies (AC1), abstraction from the details of the access control models available in different CSP platforms (AC2), portability across different CSPs (AC3), and extensibility to enforce complex authorization constraints (AC4). An in-depth discussion of how the proposed approach satisfies all the requirements in Tab. 1 is presented in Sec. 5 with particular attention to (AC5), (AC6), and (AC7). Here, we introduce the main idea underlying our approach.

While most ACaaS frameworks (e.g., [1, 10]) outsource activities pertaining to policy specification, management, and evaluation, we follow [12] and choose to outsource only policy specification and management while reusing the policy evaluation mechanism provided by the various CSPs. We do this by translating from the high-level policy specification language used in the ACaaS tool to the proprietary specification language of the various CSPs. Technically, we use a policy specification language with a formal semantics rooted in the ABAC framework [9] that is independent of a particular CSP platform. This allows us to reuse automated tools for the security analysis of policies to understand whether the defined policies meet designer expectations and perform automated policy analysis (see, e.g., [2, 18]). More importantly for this work, the formal semantics of the language of the ACaaS tool allows us to design a translation to the language available in a given CSP that can be readily enforced by the mechanisms provided by the CSP platform. Additionally, it is possible to argue the correctness of the translation, i.e. an authorization query is allowed by the formal semantics of the ACaaS tool if it is so by the access control system available in the CSP platform. This paves the way to the exploitation of the efficient integration of policy evaluation and enforcement mechanism available in CSP platforms (such as the combination of cloud and edge computing that are crucial, for instance, to reduce latency in IoT systems) and streamlines separation of concerns and identification of responsabilities.

We have implemented these ideas by extending the ACaaS tool SecurePG[3] [12] for cloud computing platforms to target IoT systems on top of the AWS IoT platform (Sec. 3.1). The extension of SecurePG (Sec. 4.2) results in the capabilities of specifying and enforcing more fine-grained policies with respect to AWS IoT and of enforcing policies in four different IoT architectures (Sec. 4.1)—integrating cloud and edge computing together with event-driven

[3]https://sites.google.com/view/securepg/

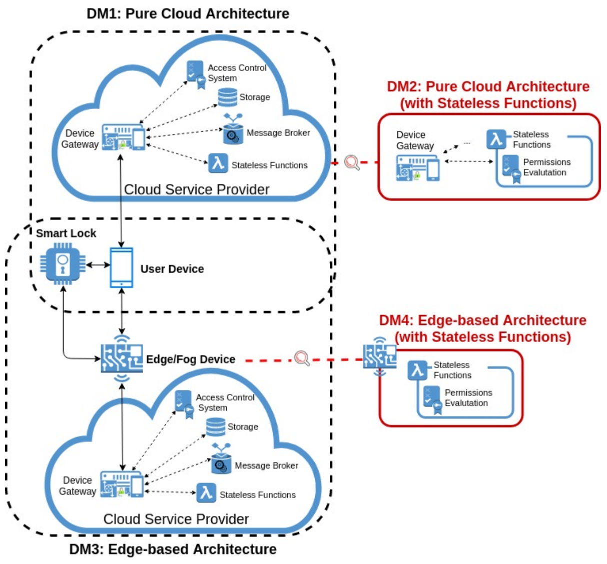

Figure 2: Four Possible Deployment Scenarios

stateless functions (Lambda function in AWS)—without additional burdens to policy designers or administrators. We show how to use the extended version of SecurePG (Sec. 4.3) to realize the smart lock system of Sec. 2.

4.1 IoT Architectures

By building upon our experience with AWS IoT and Greengrass, we identified four different architectures that exploits the capabilities of cloud and edge computing. Figure 2 provides an overview of these architectures. Red boxes highlight the delegation of policy evaluation from the native access control mechanism to stateless functions.

DM1: Pure Cloud Architecture. The access control mechanism along with IoT entities' configurations and permissions is deployed and managed in the cloud. When a user requests access to a resource, his device connects with the device gateway, which forwards the request to the native authorization mechanism for evaluation and notifies, if authorized, the device hosting the resource (e.g., a smart lock exploiting the user device Internet access).

DM2: Pure Cloud Architecture with stateless functions. As in DM1, device configurations and permissions are stored and managed in the cloud. However, this architecture extends the access control mechanism provided by CSPs through the use of cloud-hosted stateless functions (Lambda functions in AWS). Intuitively, stateless functions implement the authorization mechanism responsible for intercepting access requests and, after retrieving the needed attribute values from a database, evaluate them against the appropriate policies (eventually also fetched from the database).

DM3: Edge-based Architecture. Access control and devices management are performed in edge devices. When a user requests access to a resource, his device and the one hosting the resource interact with the edge node that evaluates the request using the native authorization mechanism (e.g., the rudimentary one based on subscriptions of AWS Greengrass).

Table 2: IoT Entities and Actions supported by SecurePG

Entities		Actions	Description
IoT Subject	Client	Connect	Support the IoT physical devices connection.
	Thing, Thing type, Things group	Subscribe, Publish, Receive	Support the IoT virtual devices subscription to Topics and, afterwards, the possibility to publish and receive messages on the Topics.
IoT Resource	Topic	Publish, Receive	Support the possibility to publish and receive messages on a Topic.
	Topic Filter	Subscribe	Support the subscription to a set of Topics.

DM4: Edge-based Architecture with stateless functions. As in DM3, device management and policy evaluation are performed locally in edge devices. However, this architecture extends the access control mechanism provided by the CSP through the use of stateless functions along the lines of DM2.

4.2 Extending SecurePG for IoT

To support the configuration of the access control mechanism in the four architectures defined in Sec. 4.1, we have extended SecurePG [12], a policy authoring framework for cloud environments that provides users with a single point of administration to (i) define authorization requirements using a CSP-independent policy specification language, (ii) automatically analyze and validate policies, (iii) translate access control requirements into platform-dependent entities and policies, and (iv) export local components and policies in pre-existing cloud environments. Next, we discuss how SecurePG has been extended to support the configuration of an arbitrary number of IoT entities with an unbounded number of attributes and their deployment in AWS IoT and Greengrass.

First, we have provided SecurePG with the capability to configure IoT entities and their primary interactions (e.g., devices connection, subscriptions to topics, publishing and receiving of messages). Table 2 presents the concepts that have been integrated in SecurePG, namely a representation of the physical devices, called *clients*, and their virtual counterpart (in the cloud), named *things*. Things can be organized in groups and possess a specific set of attributes. As in AWS, the types of subjects and resources specified by policy administrators in their policies are bound to specific actions.

To support the evaluation of policies with an arbitrary number of attributes (DM2 and DM4), we have implemented four Lambda functions, two for AWS IoT (referred to as LF1 and LF2) and two for Greengrass (referred to as LF3 and LF4). LF1 and LF3 evaluate access requests against an AWS thing's policy (i.e., its certificate's policy) and grant access if there is at least one attribute in the request that matches those in the policy and no attributes have different values. LF2 and LF4 extend LF1 and LF3 respectively by allowing the retrieval of the attributes needed for the evaluation of the request from a database. Lambda functions have been implemented in Java 8 (code available as Maven projects in the repository folder[4]) and are integrated in SecurePG. When configuring AWS IoT and Greengrass, SecurePG deploys these Lambda functions according to the selected architecture as described below; thereby completely relieving policy administrators from the burden of developing the code of Lambda functions.

Figure 3 shows the configuration and deployment procedure followed by SecurePG (red dashed rectangles denotes the functionalities developed in this work). First, users configure IoT entities

[4]Deployment models results, data and configuration: https://goo.gl/xybfGf

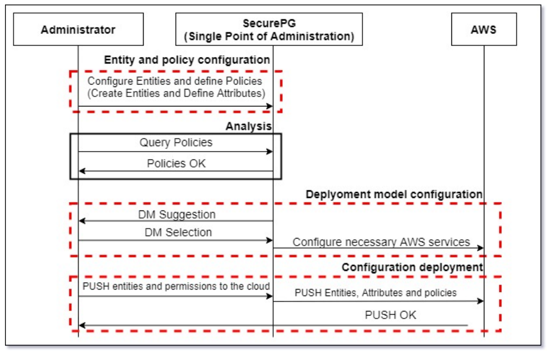
Administrator
SecurePG
(Single Point of Administration)
AWS
Entity and policy configuration
Configure Entities and define Policies
(Create Entities and Define Attributes)
Analysis
Query Policies
Policies OK
Deployment model configuration
DM Suggestion
DM Selection
Configure necessary AWS services
Configuration deployment
PUSH entities and permissions to the cloud
PUSH Entities, Attributes and policies
PUSH OK

Figure 3: SecurePG configuration and deployment procedure for IoT

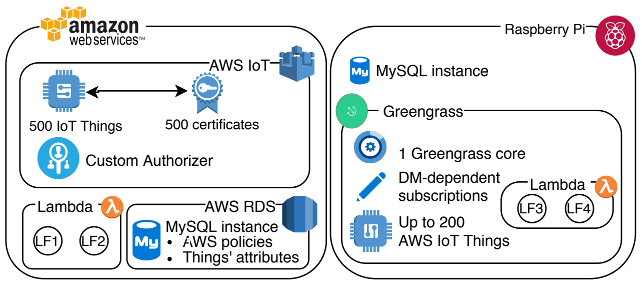

Figure 4: Prototype configuration components

and associated policies in a high level language. These policies can be analyzed and validated using a module available in SecurePG that analyzes the defined policies using an SMT solver and reports possible policy misconfigurations before their deployment (see [12] for details).

In addition, SecurePG assists users in the deployment of entities' configurations and permissions by suggesting a possible deployment based on the IoT platform and granularity of permissions. If the policy contains three or less attributes (in *Condition*), all entities are configured using AWS IoT native mechanisms (DM1). If policies contain more than three attributes, SecurePG creates and configures the AWS Lambda infrastructure to support policy evaluation using stateless functions (DM2); this requires to configure an AWS *Custom Authorizer* to coordinate policy evaluation (see Sec. 4.3). The use of attributes not supported by AWS, e.g. those associated to resources, also triggers the creation and configuration of the environment necessary to fetch policies and attributes for their evaluation.

When indicated by the user, SecurePG configures the Greengrass environment (DM3 or DM4) by triggering the deployment of the configuration on a pre-existing Greengrass-enabled device. In case policies include at least one attribute, SecurePG creates and coordinates the AWS Lambda infrastructure to support policy evaluation as in DM2 (but to be deployed on the specified Greengrass-enabled device) along with, if necessary, a local database.

When an appropriate architecture is selected, SecurePG deploys IoT entities' configurations and policies in the IoT platform. In particular, it synchronizes the necessary data (e.g., certificates) by interacting with the cloud, invokes the creation of AWS self-generated certificates and updates their local relations with the corresponding things.

4.3 The Smart lock system and SecurePG

To assess the architectures described in Section 4.1, we developed the smart lock scenario in Sec. 2 in AWS IoT and Greengrass. Figure 2 shows the interaction among connected IoT devices (smart lock and user's mobile device), AWS cloud, edge/fog device and IoT services. Entities' configurations and policies were generated and deployed using SecurePG. Here, we discuss how we did this and detail the configuration and components in Figure 4.

We simulated the interactions between smart locks and user devices by sending and receiving MQTT messages with the following software clients: JMeter[5], and a device instance of the AWS JavaScript SDK[6] via NodeJS[7] . All clients use MQTT protocol with TLS security to communicate with the AWS IoT and Greengrass services.

To implement the smart lock system using pure cloud architectures (DM1 and DM2), we configured the AWS IoT service. AWS IoT allows users to create virtual objects (things) for each physical device (using the AWS management console), create X.509 certificates (through the "one click certificate" functionality) in order to support devices interaction, and assign policies to these certificates. These certificates are attached to specific virtual objects (things) and downloaded into the corresponding IoT device. SecurePG assists policy administrators in the execution of these steps by providing a single point of administration and deploying the desired configuration on the AWS IoT platform as described in Sec. 4.2.

Additionally, for DM2 we extended the access control mechanism of AWS IoT by configuring a *Custom Authorizer* (hereafter CA). This is a special lambda function used to authorize already authenticated IoT devices by invoking LF1 or LF2. To use this function, access requests from IoT devices should contain a JSON-token and the CA signature. While the latter is verified by the AWS device gateway to authenticate the device, the token is verified by the CA with a public key (uploaded during CA's configuration).

To enable the use of LF2 in DM2, we also deployed a MySQL instance configured with *AWS Relational Database Service (RDS)*. This service provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups. The MySQL instance comprises two tables: one for JSON policies (specified using the AWS syntax) and one to store the attributes of AWS things. AWS RDS was configured with default settings.

To support edge-based solutions (DM3 and DM4), we deployed a AWS Greengrass core on a board Raspberry Pi 3 Model B and configured a Greengrass group by taking into account AWS limitations: up to 200 subscriptions and a reference to a single Greengrass core, the dedicated lambda functions (LF3 and LF4) and up to 200 AWS IoT Things' name.

[5]http://jmeter.apache.org/ - MQTT Plugin: https://github.com/emqtt/mqtt-jmeter
[6]AWS JS IoT SDK: https://github.com/aws/aws-iot-device-sdk-js
[7]https://nodejs.org

When connecting to the IoT endpoint (i.e., the Greengrass core IP address on the Raspberry Pi), clients have to provide the certificate of the Greengrass group. This has been configured with a reference to the Greengrass core, the sender and receiver IoT things, and subscription rules. The Greengrass core authenticates the connecting clients through the certificates and authorize them according to the subscription rules. These rules, however, differ between DM3 and DM4. In DM3, subscription rules have the form *<sender_name, MQTT_topic_name, "IoT Cloud">* and *<"IoT Cloud", MQTT_topic_name,receiver_name>* representing the permission to send and receive a message respectively. On the other hand, DM4 uses subscription rules of the form *<sender_name, MQTT_topic_name, Lambda_function_name>* and *<Lambda_function_name, MQTT_ topic_name, receiver_name>*. When connecting to the IoT endpoint the Greengrass core not only authenticates the client through its certificates as in DM3 but, in the case of a sender client, it also triggers the lambda function referenced in the subscription rule (the other subscription rule allows the same function to send the message to the receiver client).

Lambda functions deployed on Greengrass (LF3 and LF4) are configured as *long-lived*. Using this feature thing's attributes are cached for 30 seconds, improving the performance of policy evaluation. To support DM4 with LF4, we deployed a MySQL instance on the Raspberry Pi.

5 EVALUATION

To understand to what extent the deployment models presented in Section 4.1 meet the requirements identified in Section 2.3, we assessed the smart lock system in Section 4.3. Given the heterogeneous nature of the requirements, we evaluated them using different approaches: latency (AC5) and scalability (AC7) were evaluated by means of experiments (Sec. 5.1) while the others through a qualitative analysis (Sec. 5.2). Here we report our findings.

5.1 Experimental analysis

To evaluate the *latency* and *scalability* of the deployment models, we performed two sets of experiments using our prototype. To evaluate the latency, we configured two IoT clients using AWS IoT JavaScript SDK: one creates a timestamp when sending an MQTT message on a specific topic; the other client listens on the same topic and generates a timestamp when receiving the message. The difference between the timestamps allowed us to determine the communication and processing time for the evaluation of a IoT client's request in AWS. Clients were deployed on a 8GB ram, 8-core i7 Windows 10 machine.

The number of attributes allowed in the message (if any) and in the clients' policies (if any) varies depending on the deployment model (NodeJS Clients and configuration files available in the repository folder[4]).In DM1, the sender client is configured with a certificate that holds (in the cloud) a policy including up to three attributes. The receiver client is configured similarly with a policy that holds different attributes' values. The guarantees on clients' identity offered by assigning a certificate to each client allowed us to use clients whose policy does not contain a condition tag (i.e., no attributes are used in the policy). When connecting to the IoT endpoint (i.e., the AWS device gateway), clients provide the AWS certificate authority digital certificate, a *ClientID* (that must be equal to an AWS IoT thing's name) and its related certificate and private key. When receiving a request, the AWS IoT authorization mechanism fetches the attributes retrieved via the *ClientID* (i.e., three among the 50 specified beforehand when creating the AWS IoT thing) and evaluates their value against those retrieved via the certificate (i.e., up to three in the condition tag of the associated policy).

In DM2, the sender client is configured with a certificate that holds no condition tag; the receiver client is configured as in DM1. When connecting to the IoT endpoint, similarly to DM1, the sender client provides a *ClientID* and, instead of the certificates and key, the set of AWS CA headers to enable policy evaluation through lambda functions LF1 and LF2 as described in Section 4.3. Headers can include tokens containing up to 80 attributes due to the maximum header length supported by AWS. The lack of the certificates and keys in the requests, here and in DM4, led us to consider valid only requests sent with at least one attribute.

Since Greengrass uses a rudimentary access control mechanism based on subscriptions (cf. Sec. 3.2), we were unable to consider any policy attributes in DM3. On the other hand, lambda functions LF3 and LF4 allow the retrieval of a potentially unlimited number of attributes from MQTT messages. Thus, we tested DM4 with up to 80 attributes to compare it with DM2.

To evaluate the scalability of the deployment models, we configured a fleet of 500 policy-enabled IoT things in AWS IoT, named *test_lock_1* to *test_lock_ 500*, all belonging to a single type named *Lock*. Each thing is configured with 50 attributes, of which up to three referenced in the specific AWS IoT policies, an AWS self-generated certificate and the private key provided during its generation. Certificates hold four versions of the same policy (a sample policy for test_lock_1 is available in the repository folder[4]), enabled one at a time based on the necessary number of attributes in the condition. We assessed the deployment models by measuring the time necessary to send and receive up to 500 messages (pseudo-parallelism as supported by JMeter) from as many individual sender clients. By analysing the difference between the first timestamp (of a sender) and the last timestamp generated by the single receiver, we were able to establish the communication and processing time of all clients' messages.

In our experiments we could not test more than 500 connecting devices (499 senders and one receiver) due to the limitations of AWS IoT[8]. Moreover, the throttling on the number of parallel requests (imposed by AWS) allowed us to only evaluate policies using the attributes provided in the request (in the header or in the payload of MQTT messages). Moreover, constraints on Greengrass prevented us to evaluate more than 150 parallel requests for DM3 and 130 for DM4.

Results. We now provide a summary of the results of our experiments. Detailed results are available in the repository folder [4]. Figure 5 presents a boxplot showing the distribution of the processing time for deployment models DM1, DM2 (with LF1 for 1 to 50 attributes and with LF2 for 80 attributes) and DM4 (LF4). Lacking

[8]https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html

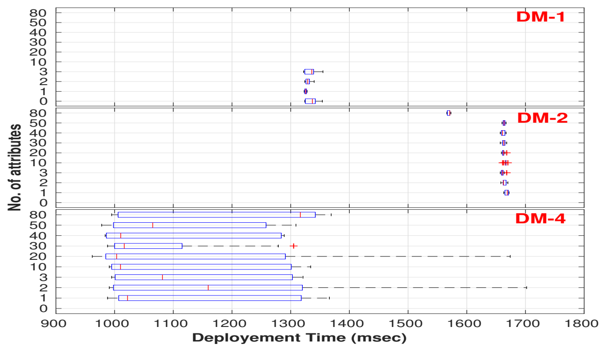

Figure 5: Distribution of time intervals (ms) to send and receive a message on a topic for the deployment models

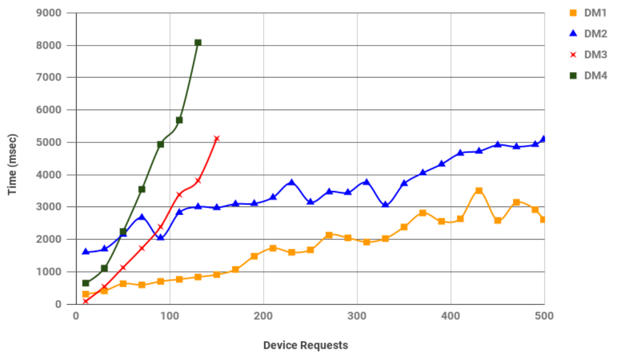

Figure 6: Time intervals (ms) necessary to send and receive multiple parallel requests

the possibility to evaluate clients' attributes in the implementation of DM3, we decided to omit this deployment model from the diagram.

From the figure, we can observe that the number of attributes specified in the policies (as in DM1), in the header (as in DM2) or in the payload (as in DM4) of the MQTT message does not significantly affect the processing time: when passing from three attributes, as supported in DM1, to the 50 attributes supported by DM2 (with LF1), the computation time varies from 1325 ms to 1668 ms. Even when DM2 executes LF2 (and connects to the AWS RDS MySQL instance to support 80 attributes), processing time ranges from 1571 ms to 1578 ms.

Our experiments show that edge-based solutions outperform pure cloud solutions for the evaluation of single requests when employing simple mechanisms (like the subscription-based mechanism provided by Greengrass and implemented in DM3) but also when interacting with a local database (as in DM4 with LF4).

It is worth mentioning that, when DM2 triggers LF1, the first client's request may take up to 4.6 times (1.9 times when DM2 executes LF2, i.e. when interacting with AWS RDS). These performance discrepancies are probably due to the caching mechanism employed by AWS to handle the container that runs lambda functions, which speeds up subsequent requests.

Figure 6 shows the results of the experiments on scalability regarding DM1, DM2 (with LF1), DM3 and DM4 (with LF4). We can observe that pure cloud architectures (DM1 and DM2) scale linearly with the number of requests, supporting up to 499 parallel requests within 2617 ms in DM1 and 5090ms in DM2. We speculate that those results are due to the scalable infrastructure offered by AWS. In contrast, we observe an exponential grow of the processing time for edge-based solutions. Our experiments also showed, as described in the full results[15], a high message loss (up to 51% when configuring 150 sender devices) for edge-based solutions.

5.2 Qualitative analysis

The prototype implementation described in Section 4.3 and our experience when assessing the four deployment models with AWS, provide important insights on how those solutions truly address the requirements identified in Section 2.3. Below we provide an in-depth discussion, which is summarized in Table 3.

DM1: Pure Cloud Architecture. This solution strongly depends on the cloud services offered by the CSP and its access control capabilities. Based on our experience with AWS, we believe that most cloud-based IoT platforms fail to fully support AC1. For instance, when handling a fleet of smart locks and user devices, system administrators would require more than three subject's attributes and, most importantly, resource's attributes to configure their policies. Similarly, the use of a proprietary, ad-hoc access control mechanisms could prevent the satisfaction of AC4 and limit the portability of access control policies (AC3). Our experiments show that AWS IoT only partially satisfies AC5 and AC7. In fact, AWS IoT processes single requests within 1.4 seconds and up to 499 parallel requests always within 4 seconds, while users of the smart lock system would expect a response time in the order of milliseconds. Moreover, we experience message loss starting from 330 parallel requests. DM1 is also not resilient in case of connection problems between users' devices and the cloud, thus not satisfying AC6.

DM2: Pure Cloud Architecture with stateless functions. Some IoT platforms like AWS IoT allow system administrators to customize and extend the provided access control mechanism, hence supporting AC1 and ACR4. This can be achieved using CSP's specific functionalities, e.g. a custom authorizer to execute LF1, or using platform independent components, e.g. a MySQL instance to fetch policies and things' attributes when executing LF2. While LF1 allows DM2 to support AC4, LF2 enables in addition the fulfillment of AC3. However, the increased empowerment corresponds to greater responsibilities for administrators. Functionalities like the recently launched AWS Custom Authorizer, which currently lacks a clear documentation, need to be fully understood to avoid design flaws that would lead to the unauthorized disclosure of sensitive resources. Our experiments show that the performances of DM2 are comparable to the ones of DM1. Moreover, we experience a message loss similar to the one we encountered in DM1 when increasing the number of (parallel) requests; thus, we mark AC6 not satisfied also for DM2.

DM3: Edge-based Architecture. As DM1, this solution is strongly bounded to the given IoT platform. However, the lack of expressibility (AC1) and extensibility (AC4) is even more evident for DM3. Edge nodes have typically reduced capabilities compared to a cloud and, thus, a simple authorization mechanism is often employed. For

Table 3: Comparison of Architectures. Green check (✔) means satisfied; orange star (✻) means partially satisfied, red cross (✗) means not satisfied.

Architectures	AC1	AC2	AC3	AC4	AC5	AC6	AC7
DM1 (AWS IoT)	✻	✗	✗	✗	✻	✗	✻
DM2	✔	✗	✻	✔	✻	✗	✻
DM3 (Greengrass)	✗	✗	✗	✗	✔	✗	✗
DM4	✔	✗	✻	✔	✔	✗	✗

instance, Greengrass uses a simple subscription-based authorization mechanism in which access rights are granted only considering the device's identifier. This approach can introduce even additional issues. For instance, we observed that, when a thing is removed from the AWS IoT service, a client was still allowed to send and receive messages using its certificates. Although single requests were processed within 500ms, due to Greengrass limitations, we experienced an exponential growth in processing time and were not able to support more than 150 parallel requests. This suggests that solutions based on DM3 can potentially satisfy AC5 but not AC7. The high message loss rate observed in our experiments suggests that additional measures have to be taken into account to address reliability (AC6).

DM4: Edge Cloud Architecture with stateless functions. By enabling ABAC through the use of stateless functions (e.g., AWS lambda functions), this solution is able to leverage the best of the pure cloud and edge-based solutions: supporting AC1 and AC4 as in DM2 and AC5 as in DM3. Our experience with AWS highlighted two important limitations that should be addressed: the evaluation of AWS IoT things' policies and the limit on the number of things that can subscribe to a given topic. In our experiments, the throttling introduced by AWS prevented us to exploit the former under heavy loads and the latter was addressed by configuring LF4 as the client sender. Although in our analysis single requests were processed within 1.5 seconds, IoT platform and edge devices' limitations might reduce DM4's potentials. For instance, within Greengrass, we experienced an exponential growth in the processing time when a single edge device was deployed. Further experiments should be performed to evaluate latency and scalability when more Greengrass cores are deployed.

Final considerations. The lack of scalability (AC7) and reliability (AC6) experienced in our experiments with DM3 and DM4, can be attributed to the way Greengrass processes MQTT messages on the Raspberry Pi, sequentially and at most 30 per second, and the language in which lambda functions have been implemented (Java). Further investigations is needed to determine if other CSPs offer edge solutions similar to Greengrass and the impact of the programming language on the results. Despite this, we believe that by employing multiple endpoints and strongly optimizing stateless functions, it is possible to efficiently handle a large number of parallel requests and to improve reliability.

It is worth noting that AWS IoT and Greengrass provide users with very little, almost no support for policy administration (AC2). To this end, we have extended SecurePG to support the semi-automated deployment of AWS Lambda functions (LF1 to LF4), the RDS configuration and all the IoT-specific components needed to manage a fleet of devices according to the specific authorization mechanism. Thus, SecurePG allows system administrators to configure and deploy their access control policies for all deployment models from a single administration point, thus satisfying AC2. Moreover, using SecurePG, administrators are not forced to specify their policies in a proprietary language. The tool offers a platform-independent language for policy specification along with components that automatically translate policy specifications into platform-specific policy specification, thus enhancing portability of policies across different CSPs (AC3).

6 RELATED WORK

Security and privacy are the primary concerns that hinder the widespread adaption of IoT. Roman et al. [14] provides an explicit analysis of the features and security challenges such as interoperability and management of access control w.r.t. various architectural approaches in the IoT. The coexistence of centralized and distributed architectural approaches is stressed for providing the foundation of full-fledged IoT. Ouaddah et al. [13] performed detailed analysis of existing access control solutions for IoT based upon domain specific IoT requirements. Alonso et al. [1] identifies special requirements (application-scoped, client-independent, flexible, delegated and configurable) that need consideration to foster new approaches for access control mechanism in IoT. Ho et al. [8] examine the security mechanisms employed in home smart lock solutions. Their study reveals flaws in the architectural design and interaction models of existing locks, which can be exploited by an adversary to learn private information about the user and gain unauthorized home access. In our work, we analyzed a vendor specific smart lock as a sample use case of IoT to devise requirements for access control to guide future development of smart locks and similar real-time IoT applications.

Access control mechanisms adopted by CSPs are typically generic and, thus, are often unable to completely capture the specific security requirements of the application domain. Moreover, they are based upon proprietary protocols leading to vendor lock-in situations, which makes the concurrent usage of different CSPs or switching between CSPs difficult for users. To address these issues, recent years have seen the emergence of several solutions adhering the principles underlying the Access Control as a Service (ACaaS) paradigm. Fitiou et al. [6] present access control as a third party service that gives data owner the flexibility to move between CSPs or concurrent usage of multiple CSP. Kaluvuri et al. [10] proposes SAFAX, a XACML-based authorization service provided by trusted third party and designed to address challenges in multi-cloud environment. It provides users with a single point of administration to specify access control policies in a standard format and augment policy evaluation with information from user selectable trust services. Alonso et al. [1] propose IoT Application-Scoped Access Control as a Service (IAACaaS) based on OAuth 2.0 protocol, an IETF standard for authorizing access to resources over HTTP that requires the resource owner to be online during the user authorizing procedure. Similarly, Fremantle et al. [7] makes use of OAuth 2.0 protocol to enable access control to information using MQTT protocol.

Outsourcing access control to trusted third party has several advantages like relieving application developers and CSPs of the burden of designing and maintaining the access control mechanism.

Moreover, it facilitates users in the configuration of their access control policies, since they can be managed from a single, central point. However, this approach is subject to the willingness of a CSP to allow the use of third party services to handle the protection of the data and resources. To the best of our knowledge, none of the existing public CSPs supports such extension. This motivated us to propose a lazy approach to ACaaS by only outsourcing activities pertaining to the specification and configuration of access control policies. This allows the definition of fine grained access control policies, employing an arbitrary number of attributes, along with dedicated function for their evaluation, which can be enforced by the native access control mechanism of the IoT platform.

7 CONCLUSION

We analyzed a realistic smart lock solution and identified the main requirements that access control systems for IoT should satisfy. Driven from this analysis and the current state-of-the-art IoT platforms, we presented an ACaaS solution that outsources the specification and administration of access control policies to a trusted third party, while leveraging the access control mechanism available in the IoT platform for policy evaluation and enforcement. We investigated the practical feasibility of the proposed approach and discussed how the identified requirements are satisfied.

Our lazy approach to ACaaS provides an initial blueprint for developing access control mechanisms for edge-cloud enabled IoT, which can be incrementally enhanced to incorporate new access control capabilities. We observed the main challenge in doing this, namely the simultaneous satisfaction of all requirements in Tab. 1 (cf. Tab. 3). The main reason for this seems to be the combination of heterogeneous technologies—such as cloud, edge and mobile computing together with communication protocols for resource constrained devices (e.g., BLE and MQTT)—that enlarge the attack surface of the access control system, hindering the possibility of confining its core functionalities to a trusted base as it is the case with more traditional systems (such as databases, operating systems, or web services). For instance, policy evaluation becomes unreliable when updates to the latest version of the policies are prevented by features of mobile computing devices such as switching to air mode in order to guarantee availability; there is an obvious trade-off between reliability (AC6) and latency (AC5). As a consequence of this state-of-affairs, it is no more possible to separate the concerns of validating and enforcing policies as typically done in the access control literature (see, e.g., [16]) that assumes that enforcement is correctly implemented by analyzing policies with respect to the abstract semantics of the specification language. Such as assumption seems be too coarse because of the subtle interactions among the technologies used in major IoT platforms. For this reason, we believe that new approaches to design and implement access control mechanisms for IoT systems must be developed and we regard this work as a first step towards this research goal. To further investigate these issues, in future work, we plan to investigate other IoT use cases and extend our ACaaS tool to support more IoT platforms.

Acknowledgments. This work is partially funded by the ITEA2 project M2MGrids (13011), the ITEA3 project APPSTACLE (15017) and the RSA-B project SeCludE.

REFERENCES

[1] Álvaro Alonso, Federico Fernández, Lourdes Marco, and Joaquín Salvachúa. 2017. IAACaaS: IoT Application-Scoped Access Control as a Service. *Future Internet* 9, 4 (2017), 64.

[2] A. Armando, S. Ranise, R. Traverso, and K. S. Wrona. 2016. SMT-based Enforcement and Analysis of NATO Content-based Protection and Release Policies. In *Proc. of the ABAC@CODASPY 2016.* 35–46.

[3] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. 2017. Access Control Model for AWS Internet of Things. In *Int. Conf. on Network and System Security.* Springer, 721–736.

[4] Charles C Byers. 2017. Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for FOG-enabled IoT networks. *IEEE Communications Magazine* 55, 8 (2017), 14–20.

[5] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security* 4, 3 (2001), 224–274.

[6] Nikos Fotiou, Apostolis Machas, George C Polyzos, and George Xylomenos. 2015. Access control as a service for the Cloud. *J. of Internet Services and Applications* 6, 1 (2015), 11.

[7] Paul Fremantle, Benjamin Aziz, Jacek Kopecký, and Philip Scott. 2014. Federated identity and access management for the Internet of Things. In *International Workshop on Secure Internet of Things.* IEEE, 10–17.

[8] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart locks: Lessons for securing commodity Internet of Things devices. In *Proc. of Asia Conf. on Computer and Communications Security.* ACM, 461–472.

[9] V. C Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J Lang, M. M Cogdell, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. 2013. Guide to ABAC Definition and Considerations. Number 800-162 in NIST.

[10] Samuel Paul Kaluvuri, Alexandru Ionut Egner, Jerry den Hartog, and Nicola Zannone. 2015. SAFAX–an extensible authorization service for cloud environments. *Frontiers in ICT* 2 (2015), 9.

[11] Nolan Mondrow. 2017. LockState 6i/6000i Update. (Aug. 2017). Retrieved Feb 13, 2018 from https://marketing.lockstate.com/acton/rif/18500/s-016e-1708/-/l-00fd:3d3/l-00fd/showPreparedMessage?cm_mmc=Act-On%20Software-_-email-_-UPDATE%20LockState%206i%2F6000i%20Issue-_-Click%20here&sid=TV2:3iibu2UNq

[12] Umberto Morelli and Silvio Ranise. 2017. Assisted Authoring, Analysis and Enforcement of Access Control Policies in the Cloud. In *IFIP International Conference on ICT Systems Security and Privacy Protection.* Springer, 296–309.

[13] Aafaf Ouaddah, Hajar Mousannif, Anas Abou Elkalam, and Abdellah Ait Ouahman. 2017. Access control in The Internet of Things: Big challenges and new opportunities. *Computer Networks* 112 (2017), 237–262.

[14] Rodrigo Roman, Jianying Zhou, and Javier Lopez. 2013. On the features and challenges of security and privacy in distributed Internet of Things. *Computer Networks* 57, 10 (2013), 2266–2279.

[15] Stavros Salonikias, Ioannis Mavridis, and Dimitris Gritzalis. 2015. Access control issues in utilizing fog computing for transport infrastructure. In *International Conference on Critical Information Infrastructures Security.* Springer, 15–26.

[16] Pierangela Samarati and Sabrina Capitani de Vimercati. 2000. Access control: Policies, models, and mechanisms. In *International School on Foundations of Security Analysis and Design.* Springer, 137–196.

[17] William Tärneberg, Vishal Chandrasekaran, and Marty Humphrey. 2016. Experiences creating a framework for smart traffic control using AWS IoT. In *Proc. of Int. Conf. on Utility and Cloud Computing.* ACM, 63–69.

[18] Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. 2017. Formal analysis of XACML policies using SMT. *Computers & Security* 66 (2017), 185–203.

[19] Xiaomin Xu, Sheng Huang, Lance Feagan, Yaoliang Chen, Yunjie Qiu, and Yu Wang. 2017. EAaaS: Edge Analytics as a Service. In *IEEE International Conference on Web Services.* IEEE, 349–356.