Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Access Control for IoT: Problems and Solutions in the Smart Home

by

Tahir Ahmad

Theses Series **DIBRIS-TH-2019-XX**

DIBRIS, Università di Genova

Via Opera Pia, 13 16145 Genova, Italy http://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Access Control for IoT: Problems and Solutions in the Smart Home

by

Tahir Ahmad

December, 2019

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
`http://www.dibris.unige.it/`

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. ING-INF/05 - Sistemi di elaborazione delle informazioni)

Submitted by Tahir Ahmad
DIBRIS, Univ. di Genova and Security & Trust Unit, Fondazione Bruno Kessler, Trento, Italy
`tahir.ahmad@dibris.unige.it, ahmad@fbk.eu`

Date of submission: December 2019

Title: Access Control for IoT: Problems and Solutions in the Smart Home

Advisor:
Silio Ranise
Researcher, Head of Security & Trust Unit, Fondazione Bruno Kessler, Trento, Italy
`ranise@fbk.eu`

Co-Advisor:
Alessandro Armando
Professor, DIBRIS, Università degli Studi di Genova, Italy
`alessandro.armando@unige.it`

Ext. Reviewers:
Luca Viganò
Professor, Head of Cybersecurity Group, King's College London, England, United Kingdom
`luca.vigano@kcl.ac.uk`

Joaquin Garcia-Alfaro
Professor, Institut Mines-Telecom & Institut Polytechnique de Paris, France
`joaquin.garcia_alfaro@telecom-sudparis.eu`

Dedication

This thesis work is dedicated to my wife, Dr. Naila, who has been a constant source of support and encouragement during the challenges of career and life. Her determination has taught me to work hard for the things that I aspire to achieve. This work is also dedicated to my parents who have always loved me unconditionally.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Silvio Ranise for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

I would like to thank Dr. Nicola Zannone in the Security group at Eindhoven University of Technology (TU/e), who provided me an opportunity to join their team as intern, and who gave access to the laboratory and research facilities. His critical remarks and inspirational feedbacks during the meetings greatly encouraged me to widen my research from various perspectives.

I would like to thank my colleagues (Alex, Amir, Andrea, Beniam, Giada, Majid, Marco, Roberto, Sali, Salvatore, Stefano and Umberto) at the Security and Trust Unit, which has been a sort of big family. Special thanks to my fellow labmate Umberto Morelli for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years.

Last but not least, I would like to express my deepest gratitude to my family and friends. This dissertation would not have been possible without their warm love, continued patience, and endless support.

Abstract

The Internet of Things (IoT) is receiving considerable amount of attention from both industry and academia due to the business models that it enables and the radical changes it introduced in the way people interact with technology. The widespread adaption of IoT in our everyday life generates new security and privacy challenges.

In this thesis, we focus on "access control in IoT": one of the key security services that ensures the correct functioning of the entire IoT system. We highlight the key differences with access control in traditional systems (such as databases, operating systems, or web services) and describe a set of requirements that any access control system for IoT should fulfill. We demonstrate that the requirements are adaptable to a wide range of IoT use case scenarios by validating the requirements for access control elicited when analyzing the smart lock system as sample use case from smart home scenario. We also utilize the CAP theorem for reasoning about access control systems designed for the IoT.

We introduce MQTT Security Assistant (MQTTSA), a tool that automatically detects misconfigurations in MQTT-based IoT deployments. To assist IoT system developers, MQTTSA produces a report outlining detected vulnerabilities, together with (high level) hints and code snippets to implement adequate mitigations. The effectiveness of the tool is assessed by a thorough experimental evaluation.

Then, we propose a lazy approach to Access Control as a Service (ACaaS) that allows the specification and management of policies independently of the Cloud Service Providers (CSPs) 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.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) is a wide ecosystem of interconnected services and devices that collect, exchange and process data, transforming the physical world and the way we live as a whole. It is a significant paradigm shift that offers many opportunities in different contexts ranging from home automation to manufacturing. At the same time these technologies presents challenges for security and privacy given the amount of potentially sensitive data such systems exchanged and process over long periods of time. Indeed, the backbone of the IoT is supported by a set of communication/messaging protocols that enable "Things" to exchange data among them or with servers hosted at the Edge or at the Cloud level.

While IoT is already making an impact on the global economy, market forecasts note that both IoT and the business models associated with it are not mature enough at this point [IEC16b]. It is believed (see, e.g., [IEC16a]) that the true potential of the IoT will be achieved only if the problems of today IoT solutions are solved or, at least, alleviated. The most important issues of IoT implementations include interoperability, latency, safety, security, trust, and privacy. If the problems related to guarantee these properties are not adequately addressed, Gartner predicts that by 2020, 80% of all IoT projects will fail at the implementation stage [IEC16b].

The 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. It is a combination of heterogeneous technologies, such as cloud, edge and mobile computing together with communication protocols for resource constrained devices (e.g., MQTT [Loc10]). The convergence of these technologies raises significant security and privacy concerns. To mitigate these concerns, access control plays a key role for providing controlled information sharing — a necessary condition to build privacy in 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., [SDV00]) 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) [OMEO17a]. As a result, the management and enforcement of access control policies become daunting tasks that prevent the deployment of secure and privacy-aware IoT solutions on hinder their adoption by users because of a lack of trust.

One of the most important approaches to ensure suitable levels of service to fulfill the properties above stems from cloud computing and its combination with edge computing. Edge-cloud IoT solutions can be supported by

- Private Providers—owned and maintained by the private entities such as IoT solution's manufacturer to provide customized, use-case specific services to the customers.
- Public Providers—-owned and maintained by public cloud service providers (CSPs), such as Amazon, Google, and Microsoft to offer a cornucopia of well-engineered and widely tested security mechanisms (e.g., Identity and Access management).

Both present challenges for the security and privacy of data and resources. The lack of built in security mechanisms in IoT devices, some of these challenges are discussed below (because they are resources constrained) put more responsibility on the IoT platform service provider. Below are the main challenges that lead to inadequate security posture for IoT platforms supported by private providers.

- The fact that developers focus on functionalities and time-to-market rather than analyzing the security implications of the design and implementation choices (e.g., it is frequent that IoT prototypes whose security has not been assessed, find their way to production environments).
- The lack of security warning and most importantly hints to patch potential vulnerabilities in components deployed with insecure configuration.

Similarly, the IoT platforms offered by the public providers suffer serious drawbacks.

- They offer limited support for policy administration.
- They are based on proprietary policy languages (which increase the risk of vendor lock-in) for policy specification and evaluation.
- They have limited expressiveness in specifying complex authorization conditions that depend on a multitude of resources and contextual attributes [AMRZ18b].

To improve the security of communications in IoT deployments based on platforms offered by private providers, we investigate the underlying publish and subscribe communication paradigm in particular the MQTT (Message Queuing Telemetry Transport) protocol and introduce an MQTT assessment tool, that is capable of (a) detecting potential vulnerabilities in MQTT brokers by automatically instantiating a set of attack patterns to expose known vulnerabilities and (b) returning a set of mitigation measures at different level of details from natural language descriptions to code snippets that can be cut-and-paste in actual deployment. Similarly, to alleviate the drawback of IoT platforms by public providers, 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 together with an expressive and high-level (independent of a particular CSP) policy specification language.

1.1 Contributions

The main contributions of this thesis are described below, grouped according to their thematic target:

- Requirements for access control systems in IoT—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.
- MQTTSA—An assessment tool for MQTT-based IoT deployment supported by private providers. The idea is to improve the security posture of IoT deployments and increase the security awareness of developers by providing accurate descriptions of the security issues and related mitigation measures.
- A *lazy* approach to Access Control as a Service (ACaaS) tailored to IoT solutions supported by public providers via outsourcing the administration and enforcement of access control policies to a trusted third party.
- SECUREPG extension for IoT—a policy authoring framework that allows users to configure, analyze and deploy their access control policies in public cloud service providers to assist users in their configurations.
- The Validation and Verification of ACaaS. The validity of approach in several use case scenarios of IoT and also with the help of the CAP Theorem. Similarly the verification of the approach is performed by a thorough experimental evaluation.

1.2 Thesis Structure

The outline of the thesis are as follows.

Chapter 2 -presents some basic concepts, which are instrumental to the definition of the proposed tools and the validation of the flexibility and adequacy of our approach,

1. CAP Theorem—used by distributed system designers to critique the design decisions.
2. MQTT Protocol—the most widely used IoT communication protocol
3. SECUREPG—a policy authoring framework for cloud environment
4. Amazon Web Services (AWS) IoT platform—a widely used cloud and edge IoT solution by Amazon Web Services (AWS).

Chapter 3 - analyzes a realistic smart lock solution and identifies the main requirements that access control systems for IoT should satisfy. The requirements are validated against a wide range of IoT use case scenarios.

Chapter 4 - uses the CAP theorem to investigate different possible design decisions. The CAP Theorem is typically used to identify the necessary trade-offs in the design and development of distributed systems, mainly databases and web applications. We use the CAP theorem for reasoning about access control systems designed for IoT.

Chapter 5 - presents the MQTT Security Assistant (MQTTSA)—a tool that automatically detects misconfigurations in MQTT-based IoT deployments. The Message Queuing Telemetry Transport (MQTT) protocol is one of the most widely used IoT communication protocols. To assist IoT system developers while deploying IoT solutions on platforms offered by private providers, MQTTSA produces a report outlining detected vulnerabilities, together with (high level) hints and code snippets to implement adequate mitigations.

Chapter 6 - presents a thorough experimental evaluation of MQTTSA by first considering a substantial number of MQTT brokers exposed to the Internet and then an MQTT broker under our control featuring five different configurations.

Chapter 7- proposes a *lazy* approach to ACaaS that allows the specification and management of policies independently of the provider while leveraging its enforcement mechanism. In case of IoT platforms offered by public providers, the security mechanism available in the cloud have been extended in IoT with shortcomings with respect to the management and enforcement of access control policies. Access Control as a Service (ACaaS) is emerging as a service to overcome these difficulties. We extend SECUREPG—a policy authoring framework that allows users to configure, analyze and deploy their access control policies in public cloud service providers to assist users in their configurations.

Chapter 8 - identifies the possible deployment models that exploits the capabilities of cloud and edge computing. However, selecting a deployment model is a complex task that requires answering several research questions. To answer these questions, we present a realization of the deployment models using the smart lock system introduced in Chapter 3. We present an experimental evaluation of the deployment models to assess their impact on the performance of the IoT system.

Chapter 9 -discusses related work to the access control solutions for smart lock systems and smart home applications, the possible architectural choices, analysis of access control mechanism for IoT and performance metrics to evaluate an access control mechanism for IoT.

Chapter 10 -draws some conclusions about the work done and discusses future developments.

1.3 Declaration

Most of the material in this thesis has already been published in scientific venues.

More in details, the work presented in Chap. 4 is co-authored with Silvio Ranise and Umberto Morelli and is submitted in an upcoming conference.

The work presented in Chap. 5 and 6 is inspired by a collaboration with Andrea Palmieri and Paolo Prem (University of Trento) and is based on [PPR+19], coauthored with Andrea Palmieri, Paolo Prem, Silvio Ranise and Umberto Morelli.

Finally, the work presented in Chapter 3, 7 and 8 is coauthored with Umberto Morelli, Silvio Ranise and Nicola Zannone and is currently under process for Journal publication and is based on the our already published works [AMRZ18a], [AR18].

1.4 Publications

1. Tahir Ahmad, Umberto Morelli, Silvio Ranise, Nicola Zannone, "A Lazy Approach to Access Control as a Service (ACaaS) for IoT: An AWS Case Study." In *23rd ACM Symposium on Access Control Models and Technologies (SACMAT)*, Indianapolis, USA.

2. Tahir Ahmad, Silvio Ranise, "Validating requirements of access control for cloud-edge IoT solutions (short paper)", In *International Symposium on Foundations and Practice of Security*, Montreal, Canada.

3. Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, Tahir Ahmad, "MQTTSA: A tool for automatically assisting the secure deployments of MQTT brokers.", In *The 1st IEEE Services Workshop on Cyber Security and Resilience in the Internet of Things*,Milan,Italy.

4. Tahir Ahmad, Umberto Morelli, Silvio Ranise, Nicola Zannone, “An Experimental Evaluation of Access Control Solutions for Internet of Things (IoT).”, is in progress for submission in *Journal of Network and Computer Applications*.

5. Tahir Ahmad, Silvio Ranise, Umberto Morelli, ”Deploying access control enforcement for IoT in the cloud-edge continuum with the help of the CAP Theorem.” submitted in *ACM Symposium on Access Control Models and Technologies (SACMAT)*, Barcelona, Spain.

Chapter 2

Background

In order to better understand our approach, this chapter provides an overview of main technologies and concepts we use. In Sect. 2.1 we start with CAP Theorem that is later used for reasoning about access control systems for IoT. Sect. 2.2 presents the survey on widely used implementation of publish-subscribe system in cloud-edge IoT solutions i.e. MQTT. In Sect. 2.3, we present SecurePG, a policy authoring framework for cloud environment. In this work, we have extended SecurePG to support IoT deployments. and lastly in Sect. 2.4, we look at Amazon Web Services (AWS) IoT platforms for cloud-edge IoT solutions, namely, AWS IoT and Greengrass. 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. We also discuss some limitations encountered in using AWS IoT and Greengrass.

2.1 CAP Theorem

In theoretical computer science, the CAP theorem, asserts that it is impossible for a distributed database system to simultaneously provide more than two out of the following three guarantees: Consistency—every read receives the most recent write or an error, Availability—every request receives a (non-error) response, without the guarantee that it contains the most recent write and Partition tolerance—the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes [Bre12]. Table 2.1 shows the logical relationship between consistency, availability and partition tolerance as potential trade-offs labelled as AP (Availability-Partition tolerance), CP (Consistency-Partition tolerance) and CA (Consistency-Availability). Considering the distributed nature of access control systems for IoT, partition is not really an option but rather a given, therefore, case (CA) is not realistic. The crucial choice is between consistency (case CP) or availability (case AP). Therefore many authors prefers the following formulation of CAP Theorem: if there is no network partition, a system can be both consistent and available; when a network partition occurs, a system must choose between either consistency (case CP) or availability (case AP) [Kle15].

Table 2.1: Trade-offs in CAP theorem

Case	Choice	Potential Trade-off
1	Consistency - Availability (CA)	Partition tolerance (P)
2	Consistency-Partition tolerance (CP)	Availability (A)
3	Availability - Partition tolerance (AP)	Consistency (C)

Despite theoretically the CAP theorem seems to be straightforward, its practical implications may be quite complex. In both CP and AP cases, it is difficult to achieve 100% consistency or availability [GL12]. The user is left with a choice depending upon the requirements of the use case scenario.

2.1.1 Applying the CAP Theorem

In the hope that the CAP theorem helps to gain insights in designing better access control mechanisms, we put its three properties in the context of enforcing policies in IoT systems.

Consistency means that the evaluation of an access request shall return a decision according to the most recently updated values of both the attributes and the policies. An access control system is consistent if an access request starts with the access control system in a consistent state and ends with the access control system in a consistent state. A highly consistent access control system can shift into an inconsistent state while evaluating access request but the entire transaction should be rolled back if an error occurring any stage in the process (such as retrieving attributes values or policies). The access requests are only processed when all the nodes are updated and during updates the access control system is not available to process any access request.

Availability requires that the access control system remain operational all the time, i.e. every access request gets a response regardless of the state in which any individual node in access control system is. A high availability access control system is desirable for IoT real-time applications. In such a system, sacrificing availability is not an option and an access control system is expected to be responsive in all situations. However, the availability can degrade easily due to the latency constraints of real-time IoT applications. The user of a smart lock has little patience and thus a fast response is critical. The expectations of the smart lock owner imply the need to sacrifice consistency to achieve sufficient availability and performance.

Partition Tolerance is a statement about the architecture of access control systems rather than systems themselves. If the access control system is partitioned, then the communication among the partitions is impossible. An access control system that is partition tolerant can sustain any partition that does not result in a failure of the entire access control system. This requires the

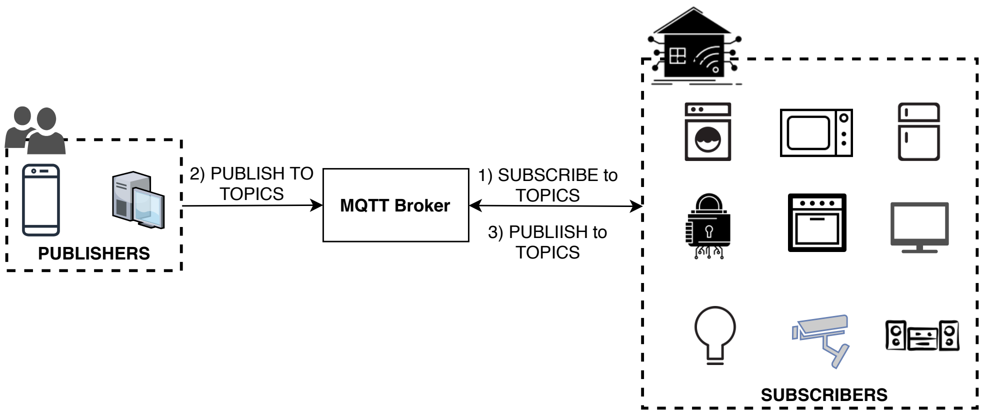


Figure 2.1: A typical smart home MQTT deployment

replication of the access control logic across combination of edge nodes and the cloud to keep the access control system up through intermittent outages. In practice, partition tolerance is seldom an option but it is a necessity.

2.2 Message Queuing Telemetry Transport (MQTT)

The publish-subscribe communication pattern follows a centralized architecture, wherein the devices or data producers publish their data to a data broker using a human-readable topic string. A typical publish/subscribe system comprises of one or more entities publishing messages, one or more entities subscribing to messages of interest, and a central mediator or message broker [RCK+19]. Applications subscribe to the data by registering their interest to a topic. The lightweight nature of publish/subscribe clients coupled with the broker's ability to isolate the publishers and subscribers in space, time, and synchronization makes production-consumption of data fully asynchronously for scalability and contrast it to client-server pattern. The implementation of publish/subscribe pattern via MQTT have been used in a broad range of IoT use cases, namely, smart homes, environmental control and building automation [SCW10].

MQTT is a lightweight messaging protocol that employs the publish-subscribe pattern: messages sent by publishing clients to transitory messaging queues—called topics—are received and (possibly) routed by a broker to topics-subscribed clients. Topics are specified by UTF8-type strings and are hierarchically organized: each level is separated by a forward slash (/) similarly to network paths and may contain two types of wildcards, namely "+" and "#". Those match, respectively, single and multiple levels in the hierarchy, and can be abused by a malicious client: if it can subscribe to the "#" topic, for instance, it will be able to receive all the messages sent by other clients. Topics specifications are used by the broker to filter messages.

Table 2.2: MQTT control messages

Control message	**Description**
CONNECT	Sent by a client to the broker, defines the type of connection to establish (parameters shown on the right)
CONNACK	Sent by the broker in response to a client CONNECT packet, it is used to acknowledge the connection request
PUBLISH	Sent by a client to the broker, provides the details and payload of the message to publish (e.g., the topic and QoS)
SUBSCRIBE	Sent by a client to the broker, provides the details related to one or more subscribing topics (e.g., QoS)
PINGREQ & PINGRESP	A mechanism to inform the broker that a clients is still connected; when receiving a PINGREQ, the broker replies with a PINGRESP.

Table 2.3: Parameters for CONNECT message

CONNECT **parameter**	**Description**
client_id	identifier of the client (can be empty)
clean_session	if false, an existing session will be used
username	optional, used for authentication
password	optional, used for authentication
will_topic	optional, topic where to publish the will message
will_qos	optional, quality of service for the will message
will_message	optional, payload of the will message
will_retain	optional, retain value for the will message
keep_alive	maximum time interval during which no messages are exchanged between a broker and a client

Figure 2.1 presents a typical smart home use case scenario (based on MQTT) to better clarify the interaction of the MQTT broker (in the middle) with user devices (on the left) acting as publishers and smart home devices (on the right) acting as subscribers. For example, a resident of the smart home can send a ”LOCK” or ”UNLOCK” values to topic home/front_door/lock on a lock installed at the front door of the smart home. The smart lock is subscribed to topic home/front_door/lock whereas the smart lock application on the smart phone can publish on the same topic i.e., home/front_door/lock.

For our work, the following facts about MQTT protocol are relevant.

- The publishers have no guarantees that messages will be delivered to intended subscribers and, similarly, subscribers have no guarantees about the identity of publishers. MQTT

Table 2.4: Return Codes

Return Code	Connection Response	Description
0	Accepted	Successfully connected
1	Refused	Unacceptable protocol version
2	Refused	Identifier rejected
3	Refused	Server unavailable
4	Refused	Bad user name or password
5	Refused	Not authorized

brokers are responsible to authenticate clients in terms of usernames and passwords or, if supported by the broker, by requesting a valid certificate (e.g., X.509) or a Pre-Shared Key (PSK).

- `CONNACK`, `PUBLISH`, and `SUBSCRIBE` include one or more `topic` names and a `quality_of_service` (QoS) that can be set to 0, 1, and 2 to indicate that messages will be sent/received by the broker at most once, at least once, and exactly once, respectively. The value of `return_code` indicates whether a connection attempt is successful or not and, in the second case, the reason of failure (see Table 2.4).For `CONNECT` messages, the `clean_session` flag tells the broker whether the client wants to establish a persistent session. If set to false, the broker will store the client session: i.e., all the subscribed topics and, in case the client subscribed with a QoS level 1 or 2, all missed messages (e.g., when the client was not connected). If set to true, the broker establishes a new session with the client: saved subscriptions and messages (if present) are deleted. The `keep_alive` parameter specifies the longest period of time that the broker and clients can endure without interacting; the parameters with prefix `will_` (or `last_will_` in some broker implementations) support the (last) "will message mechanism". This is used to notify other clients about an ungracefully disconnected client. When this happens, the broker will publish the `will_message` (set beforehand by the disconnected client) on the `will_topic` according to its QoS level `will_qos`. This message can be retained by the broker depending on the value of `will_retain`. If retained, it will be also sent by the broker to newly subscribing clients (on the `will_topic`).

- For `CONNECT` messages (see Table 2.3), the `clean_session` flag tells the broker whether the client wants to establish a persistent session. If set to false, the broker will store the client session: i.e., all the subscribed topics and, in case the client subscribed with a QoS level 1 or 2, all missed messages (e.g., when the client was not connected). If set to true, the broker establishes a new session with the client: saved subscriptions and messages (if present) are deleted. The `keep_alive` parameter specifies the longest period of time that the broker and clients can endure without interacting; the parameters with prefix `will_` (or `last_will_` in some broker implementations) support the (last) "will message mechanism". This is used

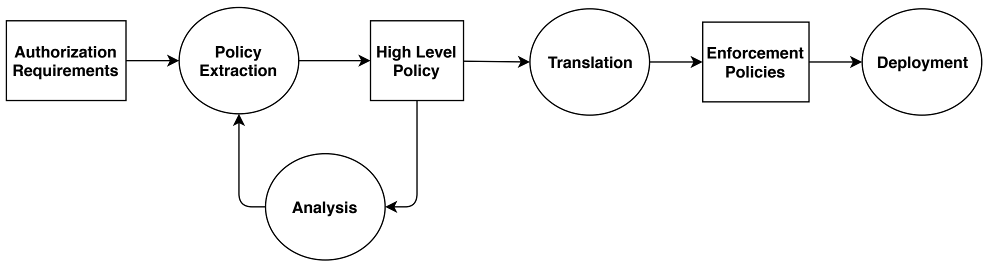


Figure 2.2: SECUREPG Workflow

to notify other clients about an ungracefully disconnected client. When this happens, the broker will publish the `will_message` (set beforehand by the disconnected client) on the `will_topic` according to its QoS level `will_qos`. This message can be retained by the broker depending on the value of `will_retain`. If retained, it will be also sent by the broker to newly subscribing clients (on the `will_topic`).

2.3 SECUREPG

SECUREPG [MR17] is a policy authoring framework for cloud environments that allows users to configure, analyze and deploy their access control policies in public cloud service providers. Figure 2.2 shows the workflow of SECUREPG, where rectangles represents artifacts and ovals represents functionalities. SECUREPG allows users to specify their *authorization requirements* in natural language, thus hiding the complexities of the particular access control model and enforcement mechanism adopted by the cloud provider. These requirements are used to extract *high-level policies* according to an abstract policy language. In addition, SECUREPG provides support to analyze and verify these high-level policies and to translate them into *enforceable policies* specified in the policy language adopted by the cloud platform. The tool also provides facilities to deploy the obtained enforceable policies in a pre-existing cloud environment.

Figure 2.3 presents the architecture of SECUREPG, which consists of three main components.

Policy Specification This component provides a graphic user interface that allows users to specify their authorization requirements in natural language. This component uses ANother Tool for Language Recognition (ANTLR) framework[1] to parse the authorization requirements and extract permissions of the form $\langle \mathit{Effect}, \mathit{Subject}, \mathit{Action}, \mathit{Resource}, \mathit{Condition} \rangle$ with the support of a general-purpose grammar. It also provides a CPR (Content Protection and Release) Translator that translates high-level policies into CPR policies based on the CPR language [ARTW16b].

Policy Analysis

[1]`https://www.antlr.org/`

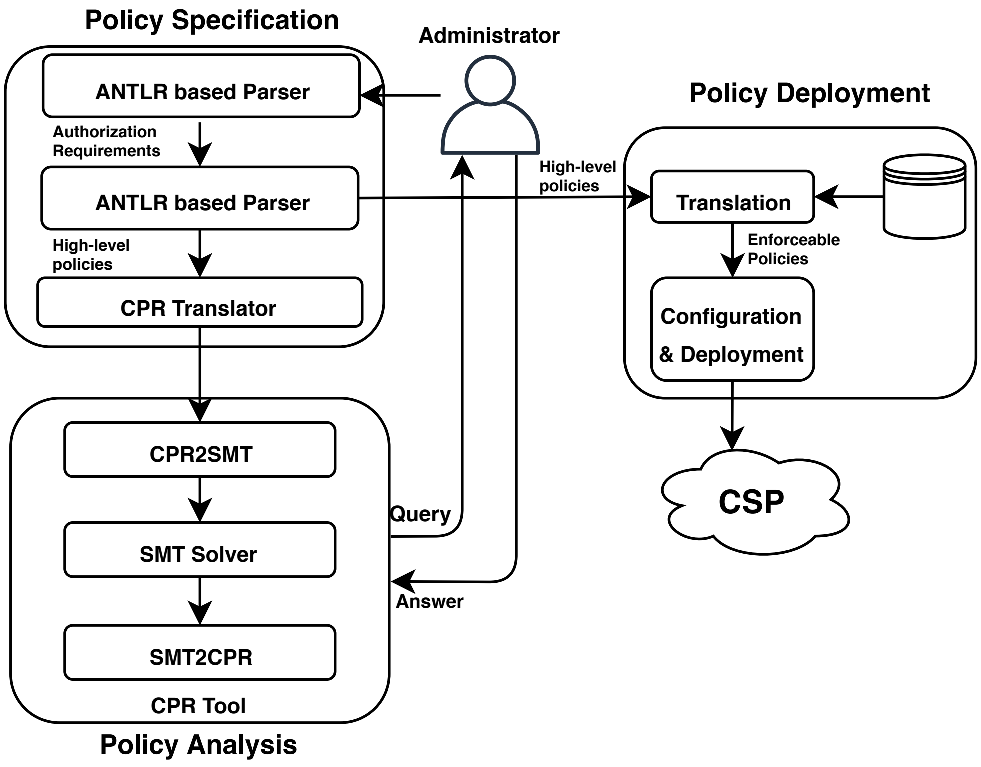


Figure 2.3: SECUREPG Architecture

This component allows users to verify the correctness of policies before their deployment. For instance, it can be used to verify whether only users with at least a certain clearance can access resources with a given level of sensitivity. Interested readers may refer to [ARTW16b] for further details.

Policy Deployment This component is responsible to deploy the specified policies into the cloud platform. It comprises a *policy translator* that converts high-level policies into enforceable platform-specific policies. The policy translator relies on a database containing a data model and the entity names used by cloud providers to instantiate high-level policies into (platform-dependent) policies that can be enforced by the cloud provider. These enforceable policies are then deployed on the selected cloud platform.

2.4 AWS IoT and Greengrass

To realize an access control mechanism for the smart lock scenario that meets the requirements for access control systems in IoT, we have investigated the use of commercial cloud and IoT

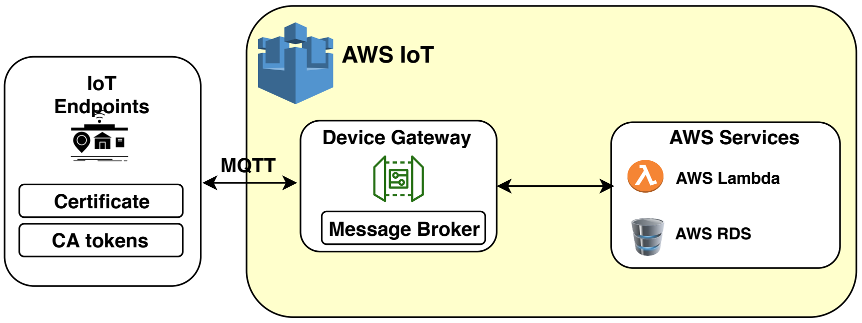


Figure 2.4: AWS IoT

platforms. In particular, we have used Amazon Web Services (AWS) and its extensions for IoT, namely AWS IoT and Greengrass, as they provide a scalable and flexible IoT infrastructure. In this section, we presents the main AWS components.

2.4.1 AWS IoT

AWS IoT is a managed cloud service that lets connected devices interact with cloud applications and other devices. It can support a large number of devices, and can process and route messages to AWS endpoints and to other devices reliably and securely. AWS IoT allows the use of AWS services like AWS Lambda and AWS Relational Database Service (RDS) to build IoT applications that gather, process, analyze and act on data generated by connected devices, without having to manage any infrastructure.

Figure 2.4 shows the main components of AWS IoT that are instrumental to the definition of the proposed access control mechanism. Below we briefly discuss these components.

- **Device Gateway** acts as an intermediary between connected devices and the cloud services, which allows these devices to talk and interact with each other. It is built in a fully managed and highly available environment in order to simplify the development of applications and provide unified security measures to all users. Secure communication between IoT devices and applications is guaranteed because messages are carried out over TLS (Transport Layer Security).

- **Message Broker** transmits messages to and from IoT devices and applications with low latency using the MQTT protocol. When communicating with AWS IoT, a client sends a message addressed to a topic like *home/front_door/lock*. The message broker, in turn, sends the messages to all the clients that have registered to receive messages from that topic.

- **AWS Lambda** provides an infrastructure to run stateless programs, called *Lambda functions*. These functions can be triggered by a wide range of sources, both internal and external to

AWS like web and mobile applications without provisioning or managing servers. It can scale instantly to hundreds of instances, with almost no platform maintenance. In this work, Lambda functions will be used to extend the native access control mechanism of AWS IoT and overcome some of its limitations.

- **Custom Authorizer (CA)** is a special Lambda function used to authenticate and/or authorize a user before Lambda functions can be invoked. The use of CA allows centralizing the authorization logic in a single function rather than packaging it up as a library into each Lambda function. When an HTTP connection is established, the Device Gateway checks if a CA is configured; if this is the case, it is used to authenticate the connection and authorize the device. The CA must return policy documents that are used by the Device Gateway to authorize MQTT operations.

- **AWS Relational Database Services (RDS)** provides a tool to set up and operate a relational database in the cloud. It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups.

AWS IoT provides an authorization mechanism to regulate access to IoT resources and devices. Any device connected to AWS IoT is authenticated through X.509 certificates. AWS IoT requires clients to provide their ID along with the corresponding X.509 certificate and checks the validity of the certificate. It then challenges the client to prove the ownership of the private key corresponding to the public key provided in the certificate. The digital certificates can be issued by a trusted third party or by AWS itself. In particular, AWS provides a "one-click certificate" functionality to generate a certificate, public key, and private key using AWS IoT's certificate authority.

Access control is enforced by mapping policies to certificates. This means that only devices or applications specified in the policies can have access to the corresponding device, which the certificate belongs to. The use of AWS policies allows a user to control access to her own devices. 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. Figure 2.5 shows an example of 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*).

```
{
    "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"}}}
    ]
}
```

Figure 2.5: AWS IoT policy for the smart lock use case

2.4.2 AWS Greengrass

To overcome potential latency issues (that are typical of pure cloud platforms for IoT), 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 the network edge to minimize the need to interact with the cloud [XHF+17].

Amazon has embraced edge computing through the AWS Greengrass service, which integrates edge computing capabilities in AWS IoT. Figure 2.6 shows the main components and services of AWS Greengrass that are instrumental to the definition of the proposed access control mechanism. Next, we discuss these components.

- **Greengrass Core** is an AWS IoT device that manages local processing and communication among IoT devices and between IoT devices and AWS IoT and AWS Greengrass services.
- **Message Broker** enables messaging between the Greengrass Core and IoT devices on the local network, facilitating communication even when there is no connection to the backend AWS platform.
- **AWS Lambda** provides an infrastructure to run Lambda functions on the AWS Greengrass core.

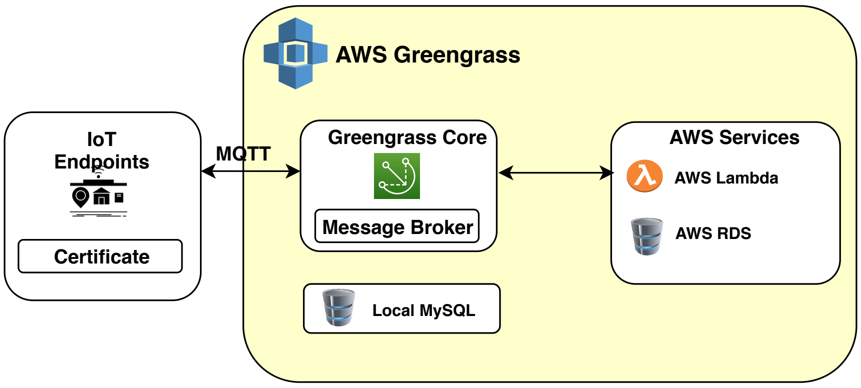


Figure 2.6: AWS Greengrass

- **AWS Greengrass Group** is a collection of settings and components, such as an AWS IoT Greengrass core, devices and subscription. Groups are used to define the scope of the interaction. For example, a group might represent the smart devices such as smart locks on one floor of a hotel.
- **Local MySQL** provides the Greengrass Core with storage capabilities.

Greengrass is equipped with a rudimentary access control mechanism. It 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.

2.4.3 Limitations of AWS IoT and Greengrass

We discuss the limitations of AWS IoT and Greengrass identified in the literature [BPS17] [AMRZ18a].

- AWS IoT and Greengrass provides little support for policy administration, every service is managed with a different administrative interface.
- AWS IoT and Greengrass does not provide any support for policy verification.
- AWS IoT does not allow the specification of fine-grained access control policies. It only allow maximum of three attributes in policy specification.
- AWS IoT is based on a proprietary access control mechanism.

- AWS Greengrass relies on a rudimentary access control mechanisms based on subscriptions, which does not allow specification of fine-grained access control polices.
- AWS Greengrass 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.

Chapter 3

Requirements of Access Control for IoT

Coordinating access to heterogeneous devices, with different needs and capabilities, is a primary concern in designing a secure IoT solution. Due to the lack of mutual compatibility between the underlying IoT platform and technologies, identifying the requirements of access control for IoT is complicated. We, therefore, performed a security analysis of a Smart Lock as a sample use case scenario of IoT to identify these requirements and guide future development of smart locks and similar IoT solutions. 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. In the end, we validate the identified requirements for other use cases of IoT and argue that the set of requirements is complete in the sense that covers all the requirements elicited in the scenarios of [SGS$^+$16].

3.1 Use Case Scenario: 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] provides smart lock solutions for residential, commercial, industrial and public buildings.

[1]https://www.amazon.com/key

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

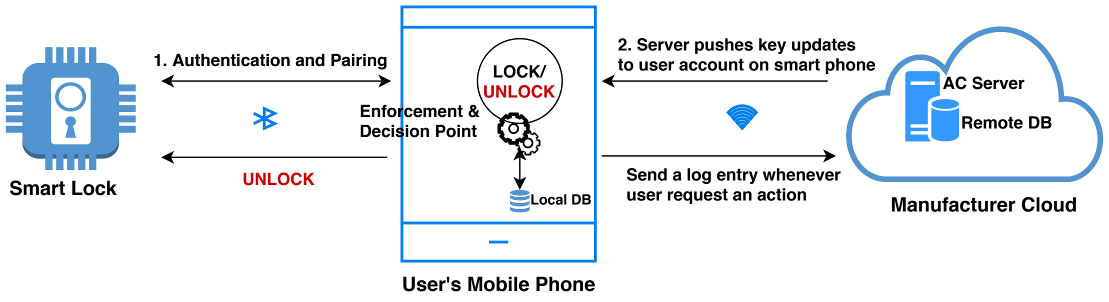


Figure 3.1: Architectural design of the smart lock system

Our use case is inspired to a vendor specific smart lock solution[3] and its representative of many other smart home solutions. 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 3.1 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) [FSG+01] 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 3.1 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 3.2, where `token_uuid` identifies the 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

[3]The name of the lock manufacturer cannot be provided due to a non-disclosure agreement.

Listing 3.1: 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 3.2: 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>
```

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 the 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 3.1, 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 maintains access by making a 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.

3.2 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.

3.2.1 Vulnerabilities

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 manufacturer'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.

3.2.2 Threat Models

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.

3.2.3 Attacks

Here, we discuss some attacks, also identified by [HLM$^+$16], 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 [Mon17].

3.2.4 Other Considerations

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 a 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 [TdRZ17]. 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

Table 3.1: 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.

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 [Bye17] 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 [FMPX15].

3.3 Requirements of Access Control for IoT

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 3.1).

(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 policies 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 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 [KEdHZ15].

(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 [SMG15]. 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 [XHF+17]. 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.

3.4 Requirements Validation

In the previous section thorough analysis of a realistic smart-lock use case scenario, we identified a minimum set of requirements that any access control system should satisfy for its effective use with cloud-edge IoT solutions. However, the smart-lock scenario is one of the many possible IoT use cases. The obvious question of the validity of the requirements naturally arises, in particular their completeness, when considering the heterogeneity of the possible use cases including transportation, health and well-being, home and building automation, smart metering, and industrial control systems. To answer the question, we consider the IoT use case scenarios in [SGS$^+$16] whose main goal is to list the relevant authorization problems of heterogeneous IoT deployments.

For the sake of brevity, we consider only two of the seven use cases from [SGS$^+$16] (however, our findings hold also for the other five use cases). Each use case description contains a table summarizing the authorization problems by using the labels used in [SGS$^+$16] for ease of reference, a high level description, and the relationship with the requirements in Table 3.1. The latter will be discussed in Section 3.5.

3.4.1 Container Monitoring for Food Transportation

Containers are used for storage and transportation of goods that need various types of climate control such as cooling and freezing. Container monitoring is a challenging task and IoT provides an opportunity for its simplification. The process involves various stakeholders such as food vendors, transporters and the super market chains, each with different monitoring requirements. The vendor packs food in sensor fitted boxes that communicate with a climate-control system. Each container carries boxes of the same owner, however, adjacent containers might contain boxes of different owners. Keeping in view, the environmental constraints on the way, the sensors might need to communicate to the endpoints over the Internet via relay stations owned by the transport company. The ownership of goods also changes on the way while they are handed over from one stakeholder to the other. The main authorization problems are the following.

- U1.1: Each stakeholder have different authorization needs of resources and endpoints.

- U1.2: Each stakeholder requires integrity and authenticity of relevant sensor data.
- U1.3: Each stakeholder requires the confidentiality of relevant sensor data.
- U1.4: Stakeholders require authorization enact without manual intervention.
- U1.5: The capability of stakeholders to grant and revoke authorization permission.
- U1.6: Ensure the reliability of authorization in presence of relay stations.
- U1.7: Ensure the reliability of authorization without access to remote authorization server.

Table 3.2: Container Monitoring use case

Authorization Problems	**Description**	**Requirements**
U1.1, U1.2, U1.3	The language used to express access control policies must ensure the integrity/ confidentiality of sensor data and also allow specification of fine-grained access control polices by different stakeholders.	AC1: Expressibility
U1.4, U1.5	Pre-configured access control that require minimal or no configuration at access time. The administration point must allow user to grant and revoke authorization permissions.	AC2: Administration
U1.2	The access control must be extensible to ensure authenticity of sensor data.	AC4: Extensibility
U1.6, U1.7	Reliability of authorization mechanism in every system state.	AC6: Reliability

3.4.2 Smart Metering

Smart meters provide a reliable and secure source for real-time insight on energy consumption. Consider an Advance Metering Infrastructure (AMI) as a use case scenario of smart metering that measures, collects, analyzes usage, and interacts with metering devices either on request or on predefined schedule. It allows consumers to control their utility consumption and aids utility providers in accurate and timely billing. Smart meters deal with sensitive user related data and are often installed in hostile locations. Which makes security assurance a concerns for users as well as service providers. The main authorization problems are the following.

- U5.1 The utility providers want to make sure that an attacker can not use data from a compromised meter to attack
- U5.2 The utility providers want to control the flow of data in their smart metering network.
- U5.3 The utility providers want to ensure the integrity and confidentiality of data.

- U5.4 Consumers want to access own usage data and also prevention of unauthorized access to such data.
- U5.5 The utility providers want the authorization policies enact even if the meters use intermediaries for Internet connectivity.
- U5.6 Authorization mechanism must be enforced during all times without human intervention.
- U5.7 Keeping in view the scale of the network, direct update of authorization policies on each and every node is almost impossible.
- U5.8 Authentication and authorization must work even if messages are stored and forward over multiple nodes.
- U5.9 Consumers want to preserve privacy by providing access to fine-grained level of consumption data to the utility providers.

Table 3.3: Smart Metering

Authorization Problems	**Description**	**Requirements**
U5.1, U5.2, U5.3, U5.4, U5.5, U5.9	The language used to express access control policies must be able to completely capture the security requirements of an organization. The access control mechanism must ensure integrity and confidentiality of user related data.	AC1: Expressibility
U5.4, U5.7	The access control system provide a single point of administration. It must allow management of user related data.	AC2: Administration
U5.5, U5.6, U5.8	Correct enforcement of authorization policies. The access control mechanism must be reliable in every system state.	AC6: Reliability
U5.7	The coherence of the access control system must be guaranteed as the network scales.	AC7: Scalability

3.5 Discussion

We argue why the authorization problems listed in Sections 3.4.1 and 3.4.2 are covered by the requirements in Table 3.1 as shown in Tables 3.2 and 3.3. This validates the requirements for the container monitoring and the smart metering scenarios. We make two observations. First, similar results can be obtained for the other five use cases in [SGS$^+$16]. Second, the only reason for which (AC3) does not show up in the analysis is that the use cases do not consider the problem of porting a solution to a different IoT platform.

(AC1): Expressibility. The language used to express access control policies must ensure the integrity and confidentiality of data and allow specification of access rules for single entity as well as group of entities. These requirements are easily satisfied by the use of a language based on ABAC which is well-known (see, e.g., [HFK^{+}13]) to support the specification of complex confidentiality and integrity goals by permitting the definition and combination of several policy idioms for defining fine-grained and context dependent authorization conditions.

(AC2): Administration and (AC5): Latency. The single point of administration is important mainly in two respects. First, it simplifies the specification of enforceable policies that result from the reconciliation of possibly conflicting security goals by different stakeholders. Support for this task comes from the precise semantics of the high-level specification language. Second, by allowing the configuration of how the enforcement of policies is performed (e.g., authorization requests are evaluated on the edge), the single point of administration permits to fine tune the system to satisfy other crucial requirements, such as Latency.

(AC4): Extensibility. To guarantee the authenticity, integrity, and confidentiality of the widely heterogeneous types of data acquired and processed by IoT devices, it is crucial to provide the access control with points of extension that allow for the integration of the most appropriate, with respect to to the type of device and use case scenario—code for data acquisition and processing. This is fundamental in the presence of constrained environments in which devices and protocols are limited and can support neither heavy computation (e.g., standard cryptography) nor communication (e.g., TLS).

(AC6): Reliability. The distributed nature of cloud-edge IoT solutions gives rise to synchronization and coherence problems that may adversely affect security; e.g., updates of access control policies should be propagated as quickly as possible to avoid making the wrong decision when the evaluation of authorization requests is distributed. To complicate the situation further is the presence of some functionality of, for example, mobile computing (such as air mode), that can be exploited to retain rights that have been revoked by presenting invalid access token to edge devices.

(AC7): Scalability. When considering very large deployments, the number of IoT devices may become so large and the topology of the network so complex to make the enforcement of evolving policies very difficult, if possible at all. An access control system for IoT should be able to blend with the elasticity of cloud-based IoT solutions to cope with a possibly exponential growth of IoT devices and the associated communication overhead.

Chapter 4

Requirements Validation Continuum with the CAP Theorem

Any tool that could help designers to understand the trade-offs involved in creating an access control system is beneficial. The CAP theorem, in particular, has been extremely useful in helping designers in understanding a wider range of systems and trade-offs [Bre12]. The CAP theorem was initially developed in the context of distributed databases and web services. However, considering the architecture of IoT deployments and their reliance on the Internet, the CAP trade-offs are still valid for IoT deployments and are even hard to tune. IoT deployments suffer from notoriously unreliable communication and considerably varying message latency; in addition, they have different properties than traditional web services. For example, security and privacy issues play an even more important role than in other types of systems as they are tightly coupled with our everyday lives. Similarly to a line of works (see, e.g., [GL12]) devoted to re-examine the CAP theorem in a wide range of technological contexts, we use the CAP theorem to better understand the unique trade-offs arising in the deployment of access control enforcement in distributed IoT systems

4.1 Access Control for IoT and running example

The main roadblock to the wider adoption of cloud-edge IoT solutions is their complexity that arises from the combination of heterogeneous techniques, including virtual machines, server-less and mobile computing together with communication protocols for resource constrained devices. This prevents the possibility of confining the core functionality of security mechanisms to a trusted base as it is the case with more traditional systems. To illustrate, consider security policy evaluation; it becomes unreliable when updates to the latest version of the policies are prevented by some features of, e.g., mobile devices such as the so called air mode that aims to guarantee availability. It is thus no more possible to separate the concerns of validating and enforcing policies as typically done in a long line of works in the literature (see, e.g., [SDV00]).

Below, we first present the three main architectures (derived from those in [AMRZ18a]) upon which enforcement mechanisms can be implemented in Cloud-Edge IoT.

4.1.1 Architectural Choices

The architecture underlying an access control mechanism has a significant impact on its performance [RLPZ19]. The cloud and IoT services offered by major public cloud service providers such as Amazon's AWS IoT and Microsoft's Azure IoT are based on policy-based access control mechanism [BPS17]. This leads us to consider policy-based access control mechanism as a reference architecture for real-world cloud-enabled IoT platforms. A widely adopted policy-based reference architecture is XACML (eXtensible Access Control Markup Language) which is the *de facto* standard for specification and enforcement of access control policies [Sta13]. It implements Attribute-Based Access Control (ABAC) that relies on attributes to identify an entity and also regulate its access by using attributes in the policies.

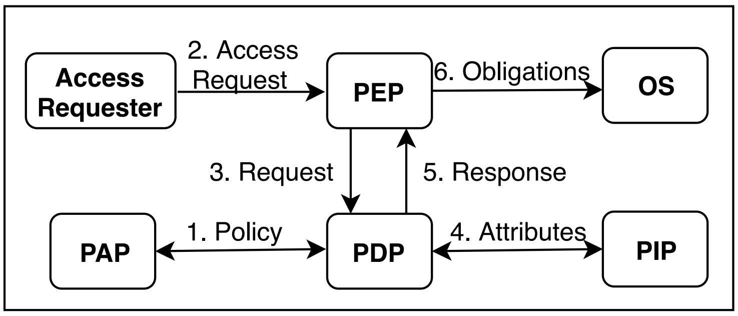


Figure 4.1: Simplified XACML Framework

Figure 4.1 shows a simplified data flow model of the XACML framework [Sta13]. We briefly explain the main components.

- The Policy Enforcement Point (PEP) intercepts authorization requests, sends them to the PDP, and waits to receive the access decision (grant or deny) possibly complemented with obligations.
- The Policy Decision Point (PDP) processes the authorization request, collects the attribute values of the requester and resource in the request (possibly also the values of contextual attributes, e.g., time of the day) by querying the Policy Information Point (PIP), evaluates the information against the policies, and produces a decision that is sent back to the PEP.

- The Policy Administration Point (PAP) allows an administrator to create and manage policies that are then made available to the PDP.
- The Policy Information Point (PIP) allows for retrieving the values of the attributes associated to the requester, the resource, and contextual information such as the time of the day or the location; it sends them to the PDP.
- The Obligation Service (OS) allows for processing directives that the PEP receives from the PDP on what must be carried out before or after an access is approved. An example is that each pair of access request and response must be timestamped and logged.

The flow for processing an access request is as follows. The PEP upon receiving an access request forwards it to the PDP that asks the PIP to retrieve the necessary attribute values so that it can evaluate the request against the policies made available by the PAP. Then, the PDP returns an access decision to the PEP together with an obligation. In our case, we only consider obligations for logging time-stamped pairs of access requests and responses.

We are now ready to present the three main architectures upon which enforcement mechanism for IoT can be implemented in Cloud-Edge computing. For the sake of concreteness, we consider a smart home scenarios in which besides an IoT device (such as a smart lock, described in more details in Chapter 3), an important role is played by a client application running on a Mobile Device (e.g., a smartphone) through which the user can interact with the IoT device. We focus on how the client application and the various entities in the XACML standard discussed above (PAP, PDP, PIP, and PEP) can be distributed among the various domains in a cloud-edge computing environment (CSP, Edge, and Mobile Device).

Table 4.1 summarizes the three possible architectures that are explained in more details in the rest of the section.

(Arch1) Cloud-based Architecture This architecture strongly depends on the cloud services offered by the CSP. The IoT endpoint connects to the back-end cloud that is hosting the PDP and PAP either via the User's Mobile Device (Arch1-) or a Hub, an Internet bridge (Arch1+) that connects the Internet with a Local Area Network created in the smart home (e.g., by a gateway router and proxy). In case of (Arch1-), the PEP is placed in the user's mobile device whereas in case of (Arch1+), the PEP resides in the Hub. In both cases, the timestamped access logs (stored in the OS of Figure 4.1) are maintained in a cloud hosted repository. In Figure 4.2, the dotted rectangle with round corners in the upper part contains both (Arch1-) and (Arch1+). AWS IoT Core—the IoT platform provided by Amazon Web Services [AWS19a]—is an example of (Arch1+) whereas the smart-lock solutions considered in [HLM^{+}16] are examples of (Arch1-); for the latter, see Section 4.1.2.

(Arch2) Edge-based Architecture Considering the reduced capabilities of edge nodes, a simplified authorization mechanism is often deployed at the network edge. In case of (Arch2), the IoT entity is directly connected to the Edge layer that is hosting (PEP), a constrained Policy Decision

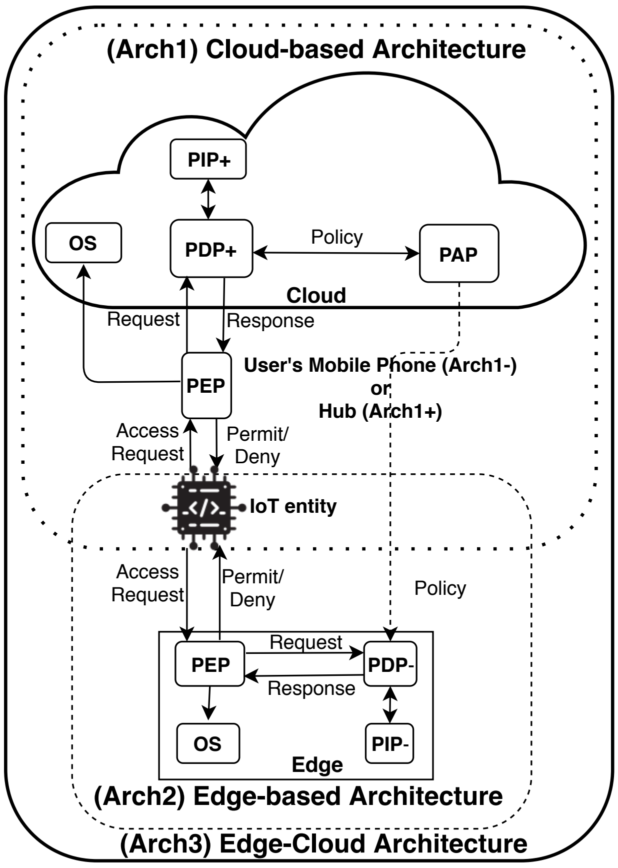


Figure 4.2: Architectures for enforcement of access control in cloud-edge based IoT

Point (PDP-) and also a constrained Policy Information Point (PIP-). For ease of administration and ensuring high availability the PAP stay in the CSP (recall requirement (AC2) in Table 3.1). Notice that access logs (in the OS) are only maintained in the Edge that becomes trusted to maintain their integrity, the crucial property to permit auditing. In Figure 4.2, the dashed rectangle with round corners in the lower part contains (Arch2). AWS IoT Greengrass—the IoT edge platform provided by Amazon Web Services [AWS19b]—is an example of (Arch2) that uses a simplified instance of ABAC based on subscriptions (roughly speaking, such a solution can be seen as a secure deployment of an MQTT broker [PPR$^+$19] based on a publish/subscribe architecture in which producers and consumers of data should subscribe to channels created by the broker).

(Arch3) Cloud-Edge Architecture Arch3 tries to combine the best of both cloud- and edge-based architectures. For instance, when real-time IoT applications are needed, the policy evaluation requires timely processing of access requests to make local processing decisions quickly and not to tolerate the latency of fetching policies from a remote storage. This paves the way to the exploitation of the efficient integration of policy evaluation and enforcement mechanism available as the combination of both edge and cloud computing that are crucial, for instance, to reduce latency of IoT systems and at the same time streamlines separation of concerns and identification of responsibilities. The constrained PDP at the Edge Node (i.e. PDP-) is backed by the fully expressive PDP in the CSP (i.e. PDP+). In Figure 4.2, the solid rectangle with round corners contain (Arch3). Notice that access logs (in the OS) are maintained both in the CSP and the Edge; in this configuration, some synchronization mechanism between the copies of the stored logs is necessary for consistency. The availability of two PDPs allow for higher flexibility in evaluating access requests; e.g., it becomes possible to enhance a simplified access control mechanism in the edge (e.g., the subscription-based one of AWS Greengrass) to a full-fledge policy-based one (e.g., that in AWS IoT Core). This is made possible by using AWS Lambda Functions [AWS19c].

4.1.2 Use Case: A Smart Lock System

Many commercial smart home lock system consist of three main components: 1) smart lock—installed on the door, 2) mobile device—to operate (lock/ unlock) the smart lock and 3) remote server—provides the management console and hosts the access control logic [HLM+16]. The users are authorised with the credentials provided by the respective smart lock manufacturer. All the access requests are logged and forwarded to the remote web server for evaluation. Upon receiving the response from the remote web service, the smart lock before enforcement update the local access matrix inside the (PEP). In case of no response from remote web server, the access decision is taken based upon the key status in the access control matrix.

The architectural design of many 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). Smart locks often lack direct Internet connectivity and, thus, they rely on the user's mobile phone or a hub (Internet bridge) to communicate with the manufacturer cloud when the phone enters the Bluetooth range of the lock [HLM+16]. Recall that Figure 4.2 shows both these possibilities as (Arch1-) and (Arch1+), respectively.

4.1.3 Relationships of CAP Theorem with access control requirements for IoT

We are now in the position to discuss the relationships between the CAP theorem and some of the most closely related requirements of Table 3.1. We also observe that the root causes underlying some violations of the requirements may be connected not only to faults but also to attackers, i.e. to security issues that will be explored further in Section 4.2.

- Latency (AC5) in IoT is influenced by many factors that range from network faults to cyber-attacks. [Aba12] argues that availability and latency are necessarily the same, i.e. an unavailable system essentially provides extremely high latency. Considering the low latency requirement for real-time IoT deployments (in terms of milliseconds), a high latency makes them unavailable. Therefore, the real trade-off is between consistency and latency. [Kle15] further asserts that availability should be modeled in terms of operation latency, i.e. defining the availability of a service as the proportion of requests that meet some latency bound; e.g., 90% of the access requests should get responses in at most 50 ms, as defined in the Service Level Agreement (SLA).
- The consistency guarantee is based on the reliability (AC6) of the underlying communication and network infrastructure in an IoT deployment. 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 a mobile device to air mode in order to guarantee availability. There is an obvious trade-off between (AC6) and availability via its deep connection with latency (AC5) as explained above. Considering IoT, designers have the option to sacrifice either availability (e.g., Best-effort availability that means to be as responsive as is possible given the current network conditions), consistency (e.g., Best-effort consistency when users require (fast) responsiveness in all situations—as it is the case of the smart lock use case—including when there are partitions and the only option is to return possibly inconsistent answers) or at times both (e.g., balancing consistency and availability). The reader interested in the details of these techniques and knowing some examples is pointed to [GL12].
- The scalability (AC7) of IoT solutions is crucial to accommodate the present needs as well as future growth. Intuitively, an IoT solution scales up if it can utilize efficiently the available resources and handle a larger workload. CAP implies a trade-off between scalability and consistency (and availability): maintaining consistency among more resources requires more communication, which in turn is subject to CAP tradeoff [GL12].

4.2 Security Analysis & the CAP Theorem

We discuss how the CAP theorem can help in mitigating security issues that may arise in IoT systems when attackers exploit vulnerabilities in access control enforcement as shown in a security analysis (Section 4.2.1). The discussion (Section 4.2.2) is structured around the relationships identified in Section 4.2.2 that allow us to characterize the trade-offs among the adoption of (Arch1), (Arch2), and (Arch3). We do this by using the smart lock scenario in Section 4.1.2 since it is representative of many smart home IoT deployments and is a good starting point to generalize our findings as demonstrated by the fact that the requirements in Table 3.1 made explicit on this scenario in [AMRZ18a] were shown to be more widely applicable in [AR18].

4.2.1 Security Analysis

Following common practice (see, e.g., [HLM$^+$16]), we structure the security analysis of (Arch1), (Arch2), and (Arch3) by describing the vulnerabilities, the threat model, and two types of attacks. For the sake of brevity and concreteness, we only consider the security issues directly related to the choice of a particular architecture when deploying the smart lock use case as a running example from Chapter 3.

Vulnerabilities: We identify the following three vulnerabilities of the enforcement mechanism that may allow users to bypass the restrictions imposed by the access control policies.

V1: The smart lock lacks direct connectivity to the Internet and relies on the local infrastructure—namely the mobile device or the Hub, see (Arch1-) and (Arch1+) respectively in Figure 4.2—to interact with the manufacturer's cloud. Thus, the smart lock implicitly trusts the local infrastructure (i.e. the user of mobile device behave faithfully or the LAN which the Internet bridge is using is not compromised).

V2: The smart lock owner can grant, update or revoke a digital key through the remote access management system. However, the smart lock receives key status updates from the remote access management system via the local infrastructure, which must not be compromised.

V3: The access control logic is implemented in the access management system hosted in the manufacturer's cloud. However, the access control policies are enforced by the (PEP) hosted in the local infrastructure. If the smart lock fails to retrieve key status updates from the remote manufacturer's cloud, the (PEP) is bound to rely on an outdated access control list.

Threat Model: We present a threat model that is typical for smart lock systems. The key observation underlying it is that attackers usually find it easier to compromise the local infrastructure that is close to the smart lock rather than the cloud service provider. We thus assume that the attacker is in *control of local infrastructure*. This can happen in two ways; either the attacker is a legitimate user of the system (Type-1) or can control the resources in the local infrastructure (e.g., mobile phone or hub) owned by a legitimate user (Type-2). For instance, a (Type-1) adversary can block the connectivity between the smart lock and the manufacturer's cloud in case of mobile device, by turning ON airplane mode when interacting with the smart lock. Instead, an example of (Type-2) adversary is a neighbor who was able to take control of the hub connecting the smart lock to the Internet by, e.g., brute-forcing weak passwords for administrators or exploiting known software vulnerabilities for which patches have not been applied.

Attacks: The adversaries can exploit vulnerabilities (V1), (V2), and (V3) performing so called *state consistency attacks*. The idea, expressed from the viewpoint of the CAP theorem, is to induce a partition so that consistency is sacrificed in favour of availability. There are mainly two variants of this idea. Below, we illustrate the attacks for (Arch1-) but similar considerations can be done for the other architectures in Figure 4.2.

Revocation Evasion (RE): A Type-1 adversary can exploit (V1), (V2), and (V3) to retain access to the smart lock when his/her permissions have been revoked by blocking the connectivity between the smart lock and the remote access management system. Consider, for instance, a housekeeper that has recently been relieved from duty. Accordingly, the owner revokes his/her permissions for entering the house. However, by switching to airplane mode on his/her mobile phone when interacting with the smart lock, he/she can prevent 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 outdated access control matrix, when it is unable to contact the access management system (V3) and so the housekeeper is still able to enter the apartment.

Logging Evasion (LE): Similar to RE, a Type-2 adversary can hide his interaction with the smart lock by blocking log messages from reaching the remote access management system by again switching to airplane mode when interacting with smart lock.

4.2.2 Architectures and Security

We now discuss the trade-offs in terms of security of deploying the smart lock use case scenario in each one of the three architectures (Arch1), (Arch2), and (Arch3) of Figure 4.2. We structure the discussion by relating requirements (AC1), (AC2), and (AC4) to the security analysis above. The only reason for which (AC3) is not considered is that we are not considering the problem of porting a solution to a different IoT platform.

Arch1: Cloud-based architecture is similar to the architecture used by most commercially used smart home lock systems. This design can be exploited by both Type-1 and Type-2 adversaries to launch both types of state consistency attacks, i.e. (RE) and (LE). The state consistency attack is possible since the list of authorized users is saved on the smartphone (Arch1-) or the local infrastructure (Arch1+) rather than a trusted component. Requirements (AC1) for expressibility and (AC3) for extensibility are satisfied because of the capabilities of the CSP and the fact that the PAP is placed in the cloud (see Figure 4.2).

Arch2: Edge-based architecture. With the addition of a trusted component (i.e., an edge Node), the smart lock should be able to keep track of users that are allowed to access the smart lock or not. Arch2 allows enforcement of administrative action (such as granting access or revoking of key) not mediated by any user intervention as it was the case in (Arch1), thus eliminating the possibility of state consistency attack for both Type-1 and Type-2 adversaries. On the negative side, the constrained nature on the part of the enforcement mechanism deployed in the edge, namely PDP- and PIP- in Figure 4.2, only allows for the specification of coarse grained access control policies. This implies that both (AC1) and (AC3) cannot be satisfied while (AC2) is still satisfied because the PAP is in the cloud.

Arch3: Cloud-Edge architecture. Similarly to (Arch2), also (Arch3) protects from state consistency attacks. At the same time, it overcomes the limitation concerning the limited expressiveness of the policies by supporting the specification of fine-grained access control policies that are desirable in IoT applications. The idea to realize this is to leverage the edge to guarantee availability when dealing with simple access control policies and resort to the CSP when more complex and fine-grained access control policies are needed. This implies that both (AC1) and (AC3) can be satisfied by allowing for a slight overhead of distributing access control requests according to the characteristics of the policies.

Finally, we observe that requirement (AC2) for administration is satisfied for (Arch2) and (Arch3) but not for (Arch1). Indeed, the PAP is always placed in the cloud (recall Figure 4.2) because administrative operations are sensitive and only the cloud can provide adequate protection mechanisms (such as multi-factor authentication and establishing secure communication channels by means of computationally expensive cryptography). However, while (Arch2) and (Arch3) offer a reasonable support for the distribution to the smart lock of updates to the policies defined in the PAP, this is not the case for (Arch1) because of the ease of partitioning the system by attackers.

4.2.3 Architectures and the CAP Theorem

We now discuss the trade-offs in terms of the CAP theorem of deploying the smart lock use case scenario in each one of the three architectures (Arch1), (Arch2), and (Arch3) of Figure 2. We structure the discussion by relating requirements (AC5), (AC6), and (AC7) to the CAP theorem as anticipated in Section 4.1.3. To contextualize the discussion, we briefly comment on the root causes of partitions that may be due to faults or to the strategy put in place by an adversary.

The smart lock, remote server and user's mobile device constitute the nodes of a distributed system. To avoid state consistency attacks such as revocation evasion, logging evasion and update evasion, the smart lock system requires a consistent access control list and access log. If partitions do not occur, when a user interacts with a smart lock using his/her mobile device, he/she relies on either a mobile phone itself or a Hub, (Arch1-) and (Arch1+) respectively, that connect the smart lock with the remote (cloud) server. In that case, the smart lock can synchronize both the access control list and the access log with the remote server (Consistency) to allow authorized lock access (Availability).

When a fault occurs or an adversary interacts with the smart lock and intercepts the communication between the smart lock and the remote server, the smart lock system suffers from a partition. To tolerate partitions, a smart lock must choose between allowing interaction with the smart lock (Availability) or rejecting requests until the smart lock can connect with the remote server and receive updates (Consistency). We notice that a partition can happen for two types of reasons: one is related to faults such as cellular outage or remote server outage and the other is the result of the malicious activity of an attacker. As a consequence, the correct choice between availability and consistency may not always be clear [HLM$^+$16].

We now discuss each architectural choice in the light of the CAP Theorem and argue which among the requirements in Table 3.1 are satisfied.

Arch1: Cloud-based architecture. The access control logic resides in the cloud, thus ensuring the reliability (AC6) of access control decisions. Due to the richness of cloud in terms of computing and storage resources, the architecture ensures scalability (AC7). However, since each request is forwarded to the (PDP) that is hosted in the cloud, this architecture is not suitable for real-time IoT deployments (that typically requires a response in few milliseconds). When an adversary interacts with the smart lock and intercept the communication between the PEP and the PDP, the system is partitioned. This gives rise to the consistency and availability trade-off. Depending on the use case requirement, it is to be decided to choose between reliability and availability.

Arch2: Edge-based architecture. To meet the latency (AC5) requirements of real-time IoT deployments, the access control logic is hosted inside an edge node closer to the smart lock system. Due to the constrained nature of an edge node, the additions of IoT devices in the deployment results in reduced reliability (AC6) of the access control mechanism. In this deployment segmentation of resources is done. All policy evaluation and enforcement components (the PDP and the PEP) are closer to the IoT endpoint, however, the PAP stays in the back-end cloud to guarantee (AC2). Due to Internet outage, the edge node might fail to interact with the back-end cloud. During partition, the service will remain available but highly unreliable (inconsistent) as any change made during that time by the administrator could not be updated on the edge node.

Arch3: Edge-Cloud architecture helps to balance consistency and availability. For example, the owner of a smart lock may specify strong consistency during day time (as no family members are home) and high availability at night time (as most of the family members are home). The work in [YV00] proposes the TACT (Tunable Availability and Consistency trade-offs) toolkit that is capable of doing this by allowing a distributed system to specify the desired level of availability and consistency. Arch3 neither guarantees strong consistency nor high availability. Even in this architecture, data can be inconsistent and a major network partitions can still render the service unavailable. Nevertheless, this architecture can significantly increase the IoT solution's robustness to partition, before compromising availability.

4.3 Discussion

We now combine the analyses of the smart lock system from the viewpoint of security (Section 4.2.2) and from the viewpoint of the CAP theorem (Section 4.2.3). Then, we present a discussion of the trade-offs underlying the satisfaction of the requirements in Table 3.1 (except for (AC3) that is outside the scope of this work). The results are presented by considering the three architectures (Arch1), (Arch2), and (Arch3) in Figure 4.2.

In the context of IoT solutions for smart homes, we are considering a CP system (with Availability trade-off) to mitigate state consistency attack. Table 4.2 shows the comparison of (Arch1), (Arch2), and (Arch3) with respect to the CAP Theorem and requirements (AC1), (AC2), (AC4)-(AC7). The main findings can be summarized as follows.

- Architecture (Arch1) fails to provide partition tolerance and (as a CP system) trades Availability (A) to achieve Consistency (C) but is vulnerable to state consistency attacks.
- Architecture (Arch2) in the presence of partition ensures availability (A) at the cost of consistency (C) by moving the access control logic (namely, the PDP and the PIP) closer to the user. However, it fails to ensure consistency (C) because the PAP is in the cloud to satisfy (AC2). On the positive side, (Arch2) makes consistency attacks a bit more difficult because adversaries need to take control of the edge node where the PEP and the OS (managing the log of pairs of access requests and responses) are running.
- The consistency (C) of an access control system in IoT (or any other network distributed systems) can be obtained by redundancy. In (Arch3), this is achieved by (partially) replicating the PEP, PDP, and PIP components of the access control enforcement both in the cloud and the edge. Architecture (Arch3) finds a balance between consistency (C) and availability (A) in the presence of network partitions (P) by evaluating requests against ”simple” policies at the edge and using the cloud for the more ”complex” ones. Additionally, (Arch3) mitigates state consistency attacks provided that adequate synchronization mechanisms for policies and attribute values are deployed between the cloud and the edge.

In the rest of the section, we provide an overview of the implications of the CAP theorem for the satisfaction of the requirement in Table 3.1 together with state consistency attacks. Our findings are summarized in Table 4.2.

Table 4.1: Architectural Choices

	Cloud	**Mobile Device**	**Edge**
(Arch1-)	PAP,PDP+,PIP+,OS	Client App, PEP	—
(Arch1+)	PAP,PDP+,PIP+,OS	Client App	PEP
(Arch2)	PAP	Client App	PEP, PDP-,PIP-,OS
(Arch3)	PAP,PDP+,PIP+,OS	Client App	PEP,PDP-,PIP-,OS

(Arch1) is able to maintain consistency (C) only in absence of partitions. Availability is also an issue because of the latency (AC5) in transmitting access control decisions from the cloud to the smart lock. Since partitions happen and can even be the result of state consistency attacks, it turns out that the administration (AC2) requirement is also not satisfied for the implied lack of synchronization between the access control logic in the cloud (namely, the PDP, PIP, and IS) and the PEP that is close to the smart lock.

Table 4.2: Comparison of Architectural Choices. (✓) means satisfied;(✚) means partially satisfied;(✗) means not satisfied.

		(Arch1-)	(Arch1+)	(Arch2)	(Arch3)
CAP Theorem	**C**	✓		✗	✚
	A	✗		✓	✚
	P	✗		✓	✓
Requirements	**AC1**	✓		✗	✓
	AC2	✗		✚	✚
	AC4	✓		✗	✓
	AC5	✗		✓	✚
	AC6	✓		✗	✚
	AC7	✓		✗	✓

(Arch2) sacrifices consistency (C) for availability (A) and partition tolerance (P) by moving a substantial part of access control enforcement closer to the smart lock. The requirements of administration (AC2) and reliability (AC6) are partially satisfied as there may be a lack of synchronization between the PAP in the cloud and the remaining components of the access control enforcement mechanism (especially the PEP) as a result of a state consistency attacks. However, such an adversarial activity is more difficult than in the case of (Arch1) because the edge (in which the PEP runs) can put in place more robust protection mechanisms. Since the edge is computationally constrained, the PDP and the PIP usually supports simpler access control policies. As a result the requirements of expressibility (AC1), extensibility (AC4), and (AC7) cannot be satisfied.

Finally, (Arch3) tries to find a compromise between consistency (C) and availability (A) in presence of partitions by replicating all the components for access control enforcement but the PAP in both the cloud and the edge. In this way, by using protocols for synchronizing the policies (in the PAP) and the logs (in the OS), (Arch3) can evaluate access requests against coarse-grained policies in the edge and invoking the components in the cloud when fine-grained policies are involved. With this strategy, the requirements for expressibility (AC1), extensibility (AC4), and scalability (AC7) are satisfied. However, (Arch3) can only partially satisfies the requirements for administration (AC2), latency (AC5), and reliability (AC6). While it is true that the strategy above can guarantee all the three requirements in absence of partitions, it is no more the case when a partition occurs especially as a result of an attack. In fact, disconnecting the cloud and the edge allows to guarantee the reliability of access control evaluation for simple policies under the assumption that no updates to such policies has been performed. Since latency constraints are also crucial in the smart lock scenario, a strategy that sacrifices consistency for availability can be adopted in those situations in which someone is likely to be at home to double check. Instead, a strategy that prefers consistency over availability can be adopted when none is at home.

Chapter 5

MQTTSA: A tool for security assessment of MQTT-based IoT deployments

MQTT—a lightweight publish subscribe messaging protocol is one of the most widely used protocol for message exchange in IoT deployments [CF18]. Major IoT service providers, such as AWS IoT, Google Cloud and Microsoft Azure IoT Hub has helped MQTT become a dominant IoT message protocol and method for enabling digital transformation. However, official MQTT specifications include no mandatory requirements for any of the typical security related aspects such as authentication, authorization, data integrity, confidentiality and the like. Following are the factors that lead to the lack of security-related functionalities in MQTT protocol [PVPG17].

- It only focuses on message dispatching.
- The protocol is not historically developed for IoT environment.
- To keep the MQTT implementation as light as possible by reducing the overhead related to security features while adopting to the constrained IoT environment.
- It is used in a very heterogeneous range of scenarios, from IoT devices to Facebook messenger mobile application, that of course require significantly different security functionalities to be rendered secure.

Similarly, the inadequate security posture of IoT systems has several root causes.

1. The lack of built-in security mechanisms and security standards for IoT devices because they are mainly resource-constrained and are rarely released with out-of-the-box security features.

2. The vast majority of IoT platforms are deployed and configured (sometimes even designed) without bearing security in mind. The developers mainly focus on functionalities and time-to-market rather than analyzing the security implications of the design and implementation choices (e.g., it is frequent that IoT prototypes whose security has not been assessed, find their way to production environments).
3. The lack of security warnings and—most importantly—hints to patch potential vulnerabilities in components deployed with insecure configurations.

While there are some work (e.g., [SRSB15]) aiming to alleviate point (1), points (2) and (3) received much less attention, especially when considering assistance to developers in mitigating well-known vulnerabilities in the context of stringent time constraints (typical of modern software production and deployment processes).

5.1 MQTT Security Assistant (MQTTSA)

To alleviate this situation and improve the security of communications in IoT deployments, we investigate the MQTT (Message Queuing Telemetry Transport) protocol and introduce a tool, called MQTT Security Assistant (MQTTSA). The ultimate goal of the tool is to improve the security posture of IoT deployments and increase the security awareness of developers with few security skills by alleviating the burden of searching and identifying the necessary information that is often scattered in several places and use different jargons; e.g., blog posts, technical and scientific papers.

Below are the capabilities of MQTTSA.

- It detects potential vulnerabilities in MQTT brokers by automatically instantiating a set of attack patterns to expose known vulnerabilities
- It returns a set of mitigation measures at different level of details—from natural language descriptions to code snippets that can be cut-and-paste in actual deployments.

MQTTSA[1] works in two steps: (Step-1) it detects potential vulnerabilities in MQTT brokers by automatically instantiating a set of attack patterns to expose known vulnerabilities; then, (Step-2) it generates a report describing a set of measures to mitigate detected vulnerabilities. The report returned by MQTTSA contains actionable descriptions of mitigation strategies at a different level of detail, either concise narratives in natural language or code snippets that can be cut-and-paste in actual deployments.

[1]https://sites.google.com/fbk.eu/mqttsa

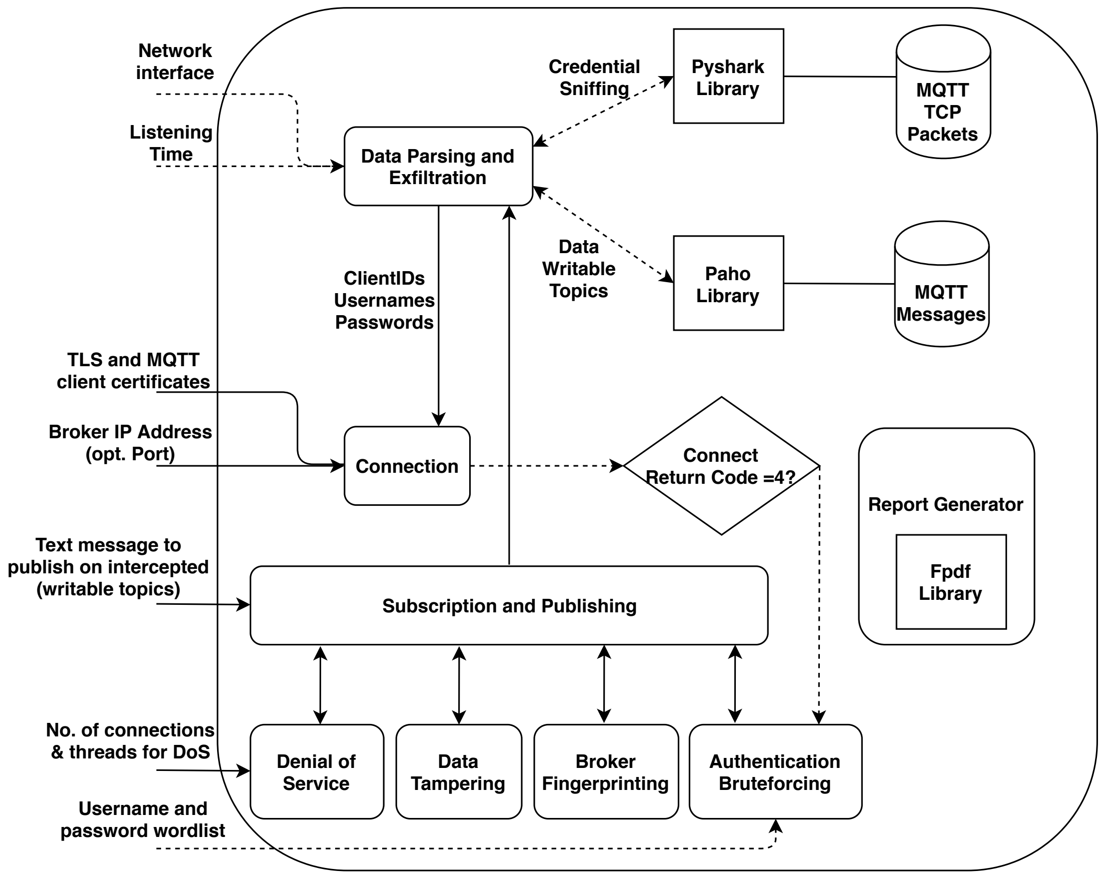

Network interface
Listening Time
Credential Sniffing
Pyshark Library
MQTT TCP Packets
Data Parsing and Exfiltration
Data Writable Topics
Paho Library
MQTT Messages
ClientIDs Usernames Passwords
TLS and MQTT client certificates
Broker IP Address (opt. Port)
Connection
Connect Return Code =4?
Report Generator
Fpdf Library
Text message to publish on intercepted (writable topics)
Subscription and Publishing
No. of connections & threads for DoS
Denial of Service
Data Tampering
Broker Fingerprinting
Authentication Bruteforcing
Username and password wordlist

Figure 5.1: The architecture of MQTTSA

The tool is developed in Python and uses two open-source libraries: Paho[2]—that provides an open-source client implementation of the MQTT messaging protocol and pyshark[3]—that provides an interface to the command line tool of Wireshark[4] (one of the most widely used packet analyzers). (Usage) Users can activate the available attack patterns by specifying options and parameters through a command line interface; the only mandatory parameter is the IP address of the MQTT broker that is going to be assessed.

5.1.1 Attack Patterns

The following attack patterns are used in this work, they are extensions to the exploits and procedures described in [Lun17, HRVL18, ARH17, FBVI17].

[2]`www.eclipse.org/paho`
[3]`kiminewt.github.io/pyshark`
[4]`www.wireshark.org`

- Data Exfiltration—combined with automated classification of potentially sensitive data.
- Data Tampering—based on a short list of test strings for fuzzing,
- Denial of Service (DoS)—performed by attempting to reduce service operation by exploiting the MQTT `will` message and the size of regular publish messages.
- Credential Sniffing—integrated with data tampering and DoS attacks.

5.2 MQTTSA Architecture

Fig. 5.1 shows the architecture of the tool whose modules are described below.

- **Connection** attempts to connect to the specified MQTT broker as a client and records the `return_code` value (recall Table 2.4). According to this and the options and parameters specified by the user, the modules below are invoked.
- **Data Parsing and Exfiltration** aims to intercept and analyse the MQTT packets exchanged between the target broker and legitimate clients in the network (with Pyshark) and on MQTT topics (with Paho). This enables MQTTSA to exploit client credentials (extracted from MQTT `CONNECT` packets) and detects the leakage of sensitive data (such as credit cards, phone numbers and emails) by using a pre-defined set of regular expressions. If the `return_code` is 0 (no authentication is required), it will disable the execution of Authentication bruteforcing as it is not needed; otherwise, this module is invoked to guess a valid password and allow the tool to connect with the broker. Notice that, to allow the interception of credentials, the user is required to set the network `interface` parameter (to, e.g., `eth0`) and run MQTTSA in the clients or broker network. If the broker implements the certificate-based authentication, the analysis can still be performed from the perspective of an insider attacker. This requires the tool to be launched with the parameters `cert` (the path to a client certificate) and `key` (path to a client private key). Then, the module attempts the subscription to the "*#*" and "*$SYS/#*" topics to intercept, respectively, messages from clients and the internal control messages of the broker.

  The internal control messages are particularly important to allow the Broker Fingerprinting module to identify the broker type and version. By default, MQTTSA records data and topics for 60 seconds (parameter `listening_time`) and includes the list of topics in the report. If the user does not enable the option "`ni`" (that restricts the tool to execute non-intrusive attacks), the module will attempt to publish a default test message (that can be modified by setting the `text_message` parameter) in each of the topics discovered with Paho; when publish succeed, the topic is added to a "writable topics" list that will be passed to the Data Tampering module.

- **Authentication bruteforcing** perform a classic password bruteforce attack in case the value of `return_code` is 4 and the tool is invoked with a username (parameter `username`) and the path to a wordlist (parameter `wordlist`). We expect this parameter to be discovered during the data parsing and exfiltration phase in non-TLS environments; provided by the tester otherwise. To the best of our knowledge, no dictionary of credentials specific to MQTT is currently available; we have thus derived one from Metasploit [Met07].

- **Data Tampering** is invoked when the user specifies the option "`md`" and the Data Parsing and Exfiltration module was able to record at least one "writable" topic (see above). This module tries to craft payloads for control packets (c.f. Table 2.2 - left) by using a pre-defined list of values with the goal of crashing the service by triggering missing input validation exceptions (with respect to the broker or the supported IoT service). It also tries to exploit specific vulnerabilities, such as CVE-2017-7650. In the Mosquitto broker, it was possible to bypass the authorisation mechanism by connecting with a wildcard username or client id, however, the vulnerability is fixed in Mosquitto version 1.4.12.

- **Denial of Service** mounts a DoS attack by publishing considerably large files for a regular IoT device (up to 10MB) and performing several concurrent requests from a single process with multiple-threads (100 by default). The idea is to evaluate the delay in the target MQTT broker upon receiving a substantial number of client requests (rather than inducing a permanent failure). Notice that clients will be disconnected if the delay exceeds the `keep_alive` value configured in the broker: this accounts for the maximum time a broker will wait before closing the connection with a non-reachable client. Interestingly, this strategy can be exploited not only to mount DoS attacks but also to spoof the credentials of an authenticated clients by forcing their disconnection and then listening for `CONNECT` packets on the network.

- **Report Generator** is invoked when all other modules have terminated their execution and it generates a report (in PDF format) collecting all the results of the attacks and a description of the mitigation measures for the detected vulnerabilities. If the **Broker fingerprinting** module was able to identify the broker type and version, the report also includes code snippets that can be readily used in the actual deployment; thereby facilitating patching. As of the current version, code snippets are generated only for Mosquitto (that resulted as the most widely used broker in the analysis of Sec. 6.1).

  Below is the example of a report resulting from the analysis of the first deployment (labelled Config1) described in the following chapter.

MQTTSA Report

Details of the assessment

Broker ip: 192.168.44.24
Listening time: 15
Text message: testtesttest
Denial of Service performed: False
Brute force performed: False

Authentication

[!] MQTTSA did not detect an authentication mechanism
The tool was able to connect to the broker without specifying any kind of credential information. This may cause remote attackers to successfully connect to the broker. It is strongly advised to support authentication via X.509 client certificates.
Moreover, it was able to intercept and use client credentials: please validate the brocker configuration.

Suggested mitigations

Please follow those guidelines and modify Mosquitto's configuration according to the official documentation. An excerpt of a configuration file is provided below:

```
listener 8883
cafile /etc/mosquitto/certs/ca.crt
certfile /etc/mosquitto/certs/hostname.crt
keyfile /etc/mosquitto/certs/hostname.key
require_certificate true
use_identity_as_username true
crlfile /etc/mosquitto/certs/ca.crl
```

Information disclosure

MQTTSA waited for 15 seconds after having subscribed to the '#' and '$SYS/#' topics. By default, clients who subscribe to the '#' topic can read to all the messages exchanged between devices and the ones subscribed to '$SYS/#' can read all the messages which includes statistics of the broker. Remote attackers could obtain specific information about the version of the broker to carry on more specific attacks or read messages exchanged by clients.

[!] MQTTSA successfully intercepted all the messages belonging to 52 topics, 1 of them non $SYS.
The non-SYS topics are: ['topic/user-due/']
The SYS topics are: ['$SYS/broker/load/bytes/sent/1min', '$SYS/broker/publish/bytes/sent', '$SYS/broker/bytes/sent', '$SYS/broker/clients/expired', '$SYS/broker/load/connections/1min', '$SYS/broker/publish/messages/sent', '$SYS/broker/load/publish/received/15min', '$SYS/broker/version', '$SYS/broker/uptime', '$SYS/broker/publish/messages/received', '$SYS/broker/messages/sent', '$SYS/broker/load/messages/sent/15min', '$SYS/broker/clients/maximum', '$SYS/broker/load/bytes/received/1min', '$SYS/broker/publish/bytes/received', '$SYS/broker/load/messages/sent/5min', '$SYS/broker/load/publish/received/1min', '$SYS/broker/clients/connected', '$SYS/broker/load/bytes/sent/5min', '$SYS/broker/load/messages/received/15min', '$SYS/broker/store/messages/bytes', '$SYS/broker/load/publish/sent/15min', '$SYS/broker/clients/active', '$SYS/broker/publish/messages/dropped', '$SYS/broker/load/sockets/15min', '$SYS/broker/load/bytes/received/15min', '$SYS/broker/clients/total', '$SYS/broker/load/publish/received/5min', '$SYS/broker/load/publish/sent/5min', '$SYS/broker/load/publish/sent/1min', '$SYS/broker/load/messages/received/5min', '$SYS/broker/messages/stored', '$SYS/broker/load/publish/dropped/15min', '$SYS/broker/clients/inactive', '$SYS/broker/load/sockets/5min', '$SYS/broker/retained messages/count',

'$SYS/broker/log/M/subscribe', '$SYS/broker/messages/received', '$SYS/broker/load/bytes/received/5min', '$SYS/broker/load/bytes/sent/15min', '$SYS/broker/load/sockets/1min', '$SYS/broker/store/messages/count', '$SYS/broker/load/connections/5min', '$SYS/broker/load/messages/received/1min', '$SYS/broker/clients/disconnected', '$SYS/broker/bytes/received', '$SYS/broker/load/publish/dropped/5min', '$SYS/broker/subscriptions/count', '$SYS/broker/load/publish/dropped/1min', '$SYS/broker/load/connections/15min', '$SYS/broker/load/messages/sent/1min']

Suggested mitigations

It is strongly recommended to enforce an authorization mechanism in order to grant the access to confidential resources only to the specified users or devices. There are two possible approaches: Access Control List (ACL) and Role-based Access Control (RBAC).
If restricting access via ACLs, please follow those guidelines and modify Mosquitto's configuration according to the official documentation. For instance, integrate the *acl_file* parameter (*acl_file /mosquitto/config/acls*) and restict a client to interact only on topics with his clientname as prefix (ACL *pattern readwrite topic/%c/#*)

Tampering data

After having successfully intercepted some messages, MQTTSA automatically created a new message (having as a payload the string 'testtesttest') and attempted sending it to every topic it was able to intercept. Remote attackers could exploit it to write in specific topics pretending to be a client (by his ID); e.g., send tampered measures to a sensor.

MQTTSA was not able to write in any topic.

Brocker Fingerprinting

MQTTSA detected the following MQTT brocker: mosquitto version 1.5.1.
[!]Mosquitto version is not updated: please refer to the last Change log for bugs and security issues.

Sniffing

MQTTSA used the specified interface to sniff the channel for 15 seconds and try to intercept credential information, such as *client-id*, *usernames* and *passwords*.

[!] MQTTSA was able to intercept credential information.
1 usernames obtained: user-uno.
1 passwords obtained: password1.
1 client-ids obtained: MQTT_FX_Client.

Suggested mitigations

We strongly suggest to enforce TLS in MQTT (secure-MQTT). TLS provides a secure communication channel between clients and server: assuming the correct configuration of TLS (secure version and cipher suites), the content of the communication cannot be read or altered by third parties.
In Mosquitto it is possible to set the *tls_version* parameter (e.g. to tlsv1.2). Refer to the official documentation for details

Warning: using MQTT over TLS could lead to a communication overhead and an increase in CPU usage, especially during the connection handshake. In devices with constrained resources, supporting TLS can have a severe impact. In these cases there are other (but less secure) solutions that could be used to secure the communication, such as encrypting only specific messages (for instance CONNECT and PUBLISH).

Additional information here:
MQTT security fundamentals: TLS / SSL

MQTT security fundamentals: how does TLS affect MQTT performance?
MQTT Security Fundamentals: MQTT Payload Encryption
MQTT Security Fundamentals: MQTT Message Data Integrity
DZone: Secure Communication With TLS and the Mosquitto Broker

Denial of service

MQTTSA was not configured or able to perform a Denial of Service attack on the broker.

Chapter 6

MQTTSA Evaluation

We measure the effectiveness of our tool by conducting an extensive experimental analysis on a large set of MQTT brokers found online (performing non-intrusive attacks to avoid service disruptions) and then running unrestricted security analysis on five deployments of a broker that are representative of as many large classes of online brokers sharing similar configurations. In the first set of experiments (Section 6.1), we perform only non-intrusive attacks (i.e. attempting a connection and data ex-filtration) to avoid data loss and service disruptions to brokers that may be part of real-world systems. In the second set (Section 6.2), instead, we run the full set of attacks available in MQTTSA on five instances of an MQTT broker—that correspond to as many large groups of configurations found online during the first set of experiments.

6.1 MQTT brokers in the wild

To discover MQTT brokers exposed to Internet we used Shodan[1], a search engine for Internet-connected devices. At the beginning of March 2019, MQTTSA was able to obtain a `return_code` between 0 and 5 (abbreviated RC below) from 40,346 online brokers as detailed in Figure 6.1; for the meaning, see Table 2.4. These results confirm the alarming trend described in [Lun17] and also highlighted in [MVQ18]: around 60% of the endpoints put message confidentiality and integrity at risk by allowing anyone to connect, publish, and receive data. We observed that, among the 10,150 brokers returning RC = 5, we were able to retrieve a server certificate only from 4 brokers; this highlights the failure to adopt TLS to secure the messages exchange. This is particularly troublesome if considering that all brokers using username and password as the authentication mechanism (i.e. those returning RC = 4) are vulnerable because passwords are transmitted as plaintext in the `CONNECT` messages (easily interceptable if listening

[1]`https://www.shodan.io`

on the network of the clients or the broker).

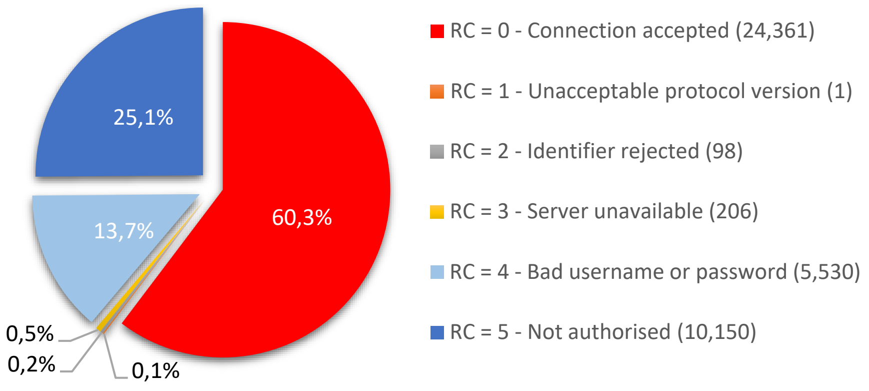


Figure 6.1: Return codes from the MQTT brokers under test

The data parsing and exfiltration module of MQTTSA (cf. Sec. 5.1) allowed us to gather additional data about the endpoints returning RC = 0. Those include statistical information on the broker (e.g., its version and the number of connected clients) and potentially sensitive data from the messages exchanged by clients. The module did this by first subscribing to all the reserved and user-defined topics (with wildcards `$SYS/#` and `#`, respectively). Then, by recording the exchanged topics and messages for 60 seconds and, finally, running a pre-defined set of regular expressions with the aim of classifying the types of messages (e.g., passwords, GPS data, MAC addresses). MQTTSA was able to intercept 2,471,590 user messages (of which 1,473,970 are unique), 803,345 user-defined topics, and 3,085,734 system messages (of which 2,999,722 are unique) from 687,004 reserved topics; the classification of the intercepted messages is shown in Table 6.1.

Table 6.1: Classification of intercepted MQTT messages automatically performed by MQTTSA

	Emails	**Passwords**	**PhoneNos**	**IoT**	**Status**	**GPS**	**API**	**IPv4**	**MAC**	**Domains**	**Directories**
Total	15,075	120,213	334	318,655	463,078	218,176	20,699	204,143	351,632	384,159	3,831,612
Uniques	1,796	59,272	132	304,987	432,905	182,298	19,043	71,472	57,771	60,666	1,147,107

It is important to observe that the lack of authentication paves the way to much more disturbing possibilities, namely, injecting messages to induce actuators to perform undesirable or out-of-context operations[MVQ18]. Since many of the messages that we have collected seem to be produced from smart home scenarios, the possibility of publishing messages can be particularly critical and may even lead to crimes as recently reported by newspapers in cases of domestic abuse[2]. We investigate this kind of possibilities from a technical point of view in the second set of experiments.

[2]`https://www.nytimes.com/2018/06/23/technology/smart-home-devices-domestic-abuse.html`

Following security concerns (also highlighted in [MVQ18]) are confirmed from the manual investigation of the messages.

- The possibility to intercept and eventually tamper with Over-the-Air upgrades (we found 44,527 messages, of which 41 relates to device firmwares) in the context of Industrial IoT.
- The leakage of smart farming telemetry (e.g., from Senzagro sensors) and healthcare sensitive data (such as patients history and various types of clinical data), references to local SQL databases (URLs, logins and passwords).
- The leakage of chat and email messages from the groupware messaging app Bizbox.

From further analysis of the available data, we were able to derive additional pieces of data. For instance, identify the version of 12,669 Mosquitto brokers (published on topic `$SYS/broker/version`): 6% between version 1.3.1 and 1.3.5; 77% between 1.4 and 1.4.9; 14% between 1.5 and 1.5.7, and only 2% the latest 1.5.8. As reported in the official changelog page[3], older versions of Mosquitto are vulnerable to trivial attacks; for instance, it is possible to perform a DoS on the versions between 1.5 to 1.5.2 as easily as publishing a message to a topic prefixed by `$` that is different from `$SYS`.[4] This test is one of the tests performed by MQTTSA.

6.2 MQTT brokers in the lab

We decided to deploy five different instances of the Mosquitto MQTT broker[5]. Those are representative of as many large groups of brokers with similar configurations that were found during the first set of experiments (Sec. 6.1). We choose Mosquitto for its widespread adoption—around 75% of the brokers that we were able to identify in the first set of experiments, i.e. that provide identifiable information in `$SYS` topics. We selected version 1.4.8 as being the most widely selected (also according to [MVQ18]). The first half of Table 6.2 (marked with **Setup**) contains the details of the five configurations used in the second set of experiments, each one deployed in a Docker container[6] on a server configured with 1 Gigabit network connection, 16GB RAM and 3.6GHz Intel Xeon E3 CPU. We observed that Mosquitto provides a mechanism to restrict access to topics based on Access Control Lists (ACLs) that is not part of the MQTT standard [BG14]. However, its implementation is quite useful to increase the security of a deployment; e.g., to prevent that malicious clients subscribe to topics and exfiltrate sensitive data.

The results of running the full set of attacks available in MQTTSA (`listenting_time` set to 60 seconds) are reported in the second half of Table 6.2 (marked with **Results**).

[3]`https://mosquitto.org/ChangeLog.txt`

[4]For details, see `https://nvd.nist.gov/vuln/detail/CVE-2018-12543`.

[5]`https://mosquitto.org`

[6]`https://www.docker.com`

Table 6.2: Features of the five deployments and results of the analysis performed by MQTTSA. (✓) means Success;(-) means Not Required ;(✚) means Partial Success;(✗) means No Success

	Setup			**Results**				
Deployment	**Protocol**	**Authentication**	**Authorization**	**Sniffing**	**Bruteforce**	**Subscribe**	**Publish**	**DoS**
Config1	TCP	None	None	-	-	✓	✓	✚
Config2	TCP	username/password	None	✓	✓	✓	✓	✚
Config3	TCP+TLS	None	None	-	-	✓	✓	✚
Config4	TCP+TLS	username/password	None	✗	✓	✓	✓	✚
Config5	TCP+TLS	certificates	ACL	✗	✗	✗	✗	✗

6.3 Discussion

Our findings (cf. Section 6.1) clearly show the possibility of exfiltrating sensitive information from a large number of deployed MQTT brokers without particular skills, as also observed in preliminary work (e.g., [HBK18, Lun17]). One of the main goals of MQTTSA is precisely to invert this troublesome trend by increasing the security awareness of IoT developers when deploying their MQTT-based solution. Below, we discuss our experience in running MQTTSA on deployments Config1–Config5 (cf. Section 6.2) and some considerations.

- **Config1** runs the default Mosquitto configuration, i.e. the broker does not support any authentication or authorization mechanism (as the majority of brokers reported in Sec. 6.1 - RC = 0), and logs its statistics on the reserved `$SYS` topics.

  Our tool was able to subscribe and publish to all the available topics (including `$SYS` ones), intercepting the messages and classifying intercepted data. The DoS attack was performed with partial success as the tool was not capable of disconnecting the clients but it was able to induce delays of one order of magnitude when connecting. Similarly, the test for malformed data (Data Tampering module) allowed MQTTSA to use unrestricted values when publishing messages (including, e.g., '\' as `client-id`), but it was not capable of exposing broker exceptions or errors. The Broker fingerprinting module was able to extract the Mosquitto version from the `$SYS` topics and highlight the use of an outdated broker. These results confirm the unacceptable security posture of the default Mosquitto configuration: anyone is able to subscribe and publish to any topic. This, combined with the possibility of sending malformed data, allows attackers to probe the broker and trigger error conditions or crash in the supported IoT service; especially if not validating received data.

- **Config2** enforces password-based authentication (as the brokers returning RC = 4 in the experiments of Sec. 6.1) with a session expiry time of 60 minutes. The broker blocks anonymous connections and, to offer improved performance, logs no data (locally or in `$SYS` topics) and limits the payload of client messages to 5MB.

MQTTSA was able to detect the presence of the authentication mechanism and tried to intercept client credentials by sniffing `CONNECT` packets (remember that credentials are transmitted unencrypted). If the tool was not able to intercept any `CONNECT` packet (since, for instance, clients were already connected) and a wordlist of passwords was passed as input, the Authentication Bruteforcing module attempted guessing valid credentials. Once obtained a set of credentials, they were used to connect, read and publish on available topics, and to perform the malformed data attack. Since MQTT does not support multiple connections from the same `client-id`, instead of (in addition) from the same username, the DoS attack was still possible although limited by the 5MB restriction on the size of the messages. When connecting with the same username and client-id, e.g. when exploiting intercepted credentials or if setting the parameter `use_username_as_clientid` of the Mosquitto configuration to *true* (enforce unique usernames), MQTTSA disconnected the legitimate client; the disconnection can be detected, for instance, by verifying the number of connected clients from the `$SYS` topics (not in Config2).

These results demonstrate that the adoption of an authentication mechanism based on username and password does not improve the security level with respect to Config1. As possible mitigations for DoS attacks, MQTTSA suggests to restrict the size of payload messages and to enforce the uniqueness of usernames (if allowed by the broker); doing so, a user may not (either inadvertently or maliciously) use his/her identity to damage the service or crash the broker; e.g., by publishing multiple increasingly large messages.

- **Config3** extends Config1 with TLS to encrypt the communication between the broker and clients. Unfortunately, since the use of TLS is not combined with an authorisation mechanism, it is still possible for anyone to publish and subscribe to available topics. As a consequence, the attacks described for Config1 are still possible. Our tool was able to confirm these observations.

- **Config4** extends Config2 with TLS. MQTTSA was able to detect the presence of the password-based authentication mechanism and automatically performed the authentication bruteforcing and the credential sniffing attacks: only the former was effective due to the non-possibility to extract credentials from encrypted `CONNECT` packets. However, once guessing a valid set of credentials, the tool was able to connect and interact with the broker, thereby confirming the possibility of mounting the same attacks discussed in Config2. Similar to Config3, in Config4 any authenticated client is able to publish and subscribe (unrestrictedly) on any topic.

  These results show that the combination of TLS with password-based authentication does not provide an adequate level of security in MQTT deployments.

- **Config5** implements all the suggested security mechanisms, ranging from the use of TLS with (X.509) certificate-based authentication to ACL-based restrictions on topics. None of the attacks available in MQTTSA was successful for this deployment unless running the tool with a valid client certificate: we support this feature to test the broker configuration against insider attackers (verifying, for instance, the resilience of the IoT service towards malformed data or DoS attacks) and help understand the effectiveness of implemented ACLs and the sensitivity of messages the tool was able to intercept.

While these results highlight that it is possible to secure an MQTT broker deployment with a combination of security mechanisms, we are aware that this is not always feasible due to, for instance, resource-constrained devices or specific use cases (e.g., latency-sensitive ones). For this reason, we strongly believe that research in developing security mechanisms that are less resource-intensive (e.g. [SRSB15]) is becoming of paramount importance. In particularly hostile environments, we suggest (in addition) to complement the use of MQTTSA with a complete fuzz testing (see, e.g., [FS15]) of the IoT service and possibly a firewall and a load balancer (see, e.g., [CF18]).

- As a final remark, we run MQTT-PWN [AZ18] on Config1-Config5 to compare the tool capabilities: MQTT-PWN was able to perform only data exfiltration in Config1 and authentication bruteforcing in Config2. It was not possible to exfiltrate data from Config2 (as an authenticated tester) and assess Config3-Config5 due to the implementation of TLS (and password- or certificate-based authentication). Finally, MQTT-PWN is not able to provide a report and the hints on possible mitigations as MQTTSA.

Chapter 7

A Lazy Approach to Access Control as Service (ACaaS)

The security concerns are one of the factors that impede the widespread adoption of IoT technology. The security mechanisms offered by major cloud services providers are based on well-engineered but generic, complex and proprietary access control mechanisms. Therefore, adaption process requires the creation of complex protocols, provides minimal control to end-users over outsourced data and often leading to complex problems and vendors ”lock-in” conditions [FMPX15].

In this chapter, we present an access control solution based on Access Control as a Service (ACaaS)—a cloud computing paradigm that is based on the outsourcing of access control activities to a trusted third party. ACaaS eliminates the need of developing complex adaptation protocols, enhances end-users privacy and offers data owner the flexibility to switch among service providers. to configure devices and specify access control policies with the interfaces provided by the (ACaaS) service provider [FMPX15].

We first identify the limitations of existing IoT platforms, in particular, AWS IoT platform. Then we propose a lazy approach to Access Control as a Service (ACaaS) and explain how to design an effective access control mechanism that fulfills the requirements of access control for IoT and has the ability to overcome the limitations of the AWS IoT platform.

7.1 Limitations of AWS IoT & Greengrass

In the following we will discuss the limitations encountered while using AWS IoT and Greengrass to implement the smart lock system described in Chapter 3 with respect to the requirements listed in Table 3.1:

- AWS IoT does not allow the specification of fine-grained access control policies (AC1). The dynamic nature of IoT might demand to express complex authorization conditions involving a large number of attributes and/or refer to properties of the requested resource and environment [HGP+18]. In contrast, AWS IoT restricts the number of attributes that can be used for policy specification to three, resulting in a too coarse grained access control for dynamic environments.
- AWS IoT provides limited support for policy administration (AC2). Every service has a different administrative interface, 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 verify the correctness of 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, thus hindering the migration to other IoT platforms and resulting in vendor lock-in (AC3).
- 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).
- 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.
- To the best of our knowledge, AWS IoT has only been tested with small scale deployments whereas large scale deployments (AC7) with different sets of requirements as the ones given in Chapter 3, are still unclear [TCH16].

7.2 A Lazy Approach to ACaaS

First of all, we observe that the first four requirements in Tab. 3.1 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 [HFK+13], 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. 3.1 is presented in Sec. 4.2 with particular attention to (AC5), (AC6), and (AC7). Here, we introduce the main idea underlying our approach.

While most ACaaS frameworks (e.g., [AFMS17, KEdHZ15]) outsource activities pertaining to policy specification, management, and evaluation, we follow [MR17] 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 [HFK$^+$13] 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., [ARTW16a, TdRZ17]).

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 responsibilities.

7.3 Deployment Models for Cloud-Edge IoT solutions

Based on the architectural choices (cf. Chapter 4) and by building upon our experience with AWS IoT and Greengrass, we identified seven different deployments that exploits the capabilities of cloud-edge IoT. AWS IoT Core allows the possibility to deploy (Arch1) cloud-based architecture, whereas AWS Greengrass allows the possibility to deploy both (Arch2) edge-based architecture and (Arch3) cloud-edge architecture. In case of (Arch2), AWS Greengrass transfer messages between clients on QoS-0, whereas for (Arch3), the messages are routed via back-end cloud that supports QoS-1. Thus both edge involving architectures gives the flexibility to choose between high availability or reliability. The deployment models in our work are mainly based on (Arch1) and (Arch2) architectural choices.

Figure 7.1 provides an overview of these deployments. Red boxes highlight the delegation of policy evaluation from the native access control mechanism to stateless functions.

DM1: Cloud-based Deployment This architecture resembles the access control mechanism provided by AWS IoT (Section 2.4.1). The access control mechanism along with IoT entities' configurations and permissions is deployed and managed in the cloud. When a user requests

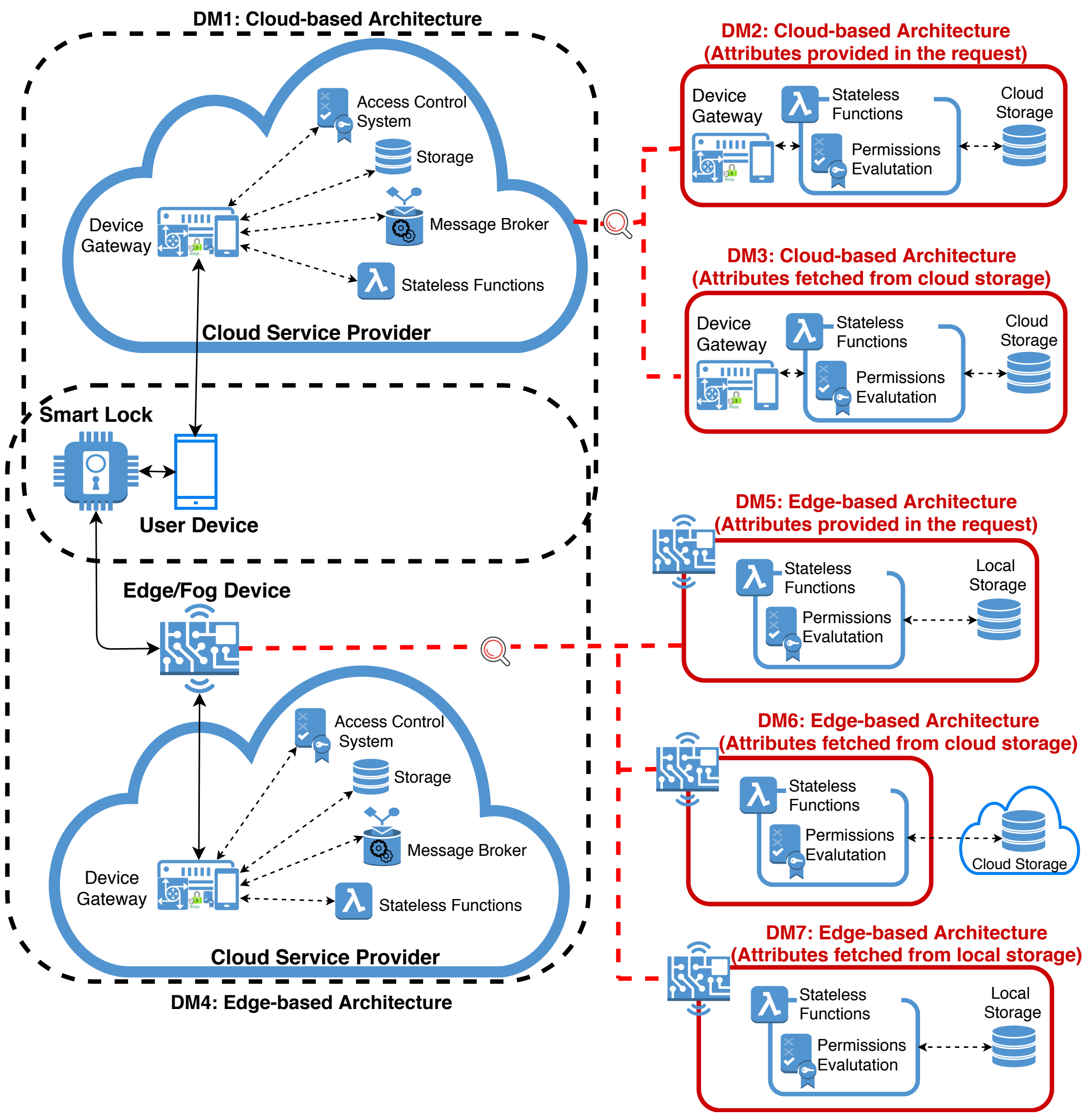


Figure 7.1: Seven Possible Deployment Models

access to a resource, his device connects with the device gateway (located in the cloud), which forwards the request to the authorization mechanism for evaluation.

DM2: Cloud-based Deployment with attributes in access request As in DM1, device configurations and permissions are stored and managed in the cloud. However, this architecture extends the access control mechanism provided by the IoT platform through the use of stateless functions (Lambda function in AWS IoT). When a user requests access to a resource, the request is intercepted by the device gateway, which forwards the request to the stateless function. The stateless function retrieves attributes provided in the access request and the necessary policies needed for policy evaluation are fetched from a cloud-based storage.

DM3: Cloud-based Deployment with Cloud Storage As for DM2, DM3 relies on a stateless function for policy evaluation. However, DM3 differs from DM2 in the way the attributes required for policy evaluation are provided to the access control mechanism. Specifically, attributes are stored and retrieved from a repository deployed in the cloud. Once an access request is forwarded to the stateless function, the function retrieves the necessary attributes from the repository and uses them to make an authorization decision.

DM4: Edge-based Deployment The incapability of cloud-based architecture to meet the requirements of latency sensitive IoT applications has raised the need for a flexible multilevel architecture in which heterogeneous devices at the edge of the network collect data, compute tasks with minimal latency, and produce localized actions [PAG$^+$18]. DM4 follows this computing paradigm such that the access control logic along with devices' configurations and permissions is deployed and managed on the edge. This architecture resembles the native access control mechanism provided by Greengrass (Section 2.4.2). When a user requests access to a resource, his device and the one hosting the resource interact with the edge node, which evaluates the request using the native access control mechanism (a subscription table in Greengrass).

DM5: Edge-based Deployment with Attributes in Access Request DM5 extends DM4 through the use of stateless functions acting in the edge node. When the edge device receives an access request, it triggers a stateless function for its evaluation against the defined policies. The stateless function retrieves the attributes from the access request and uses those attributes to evaluate the access request.

DM6: Edge-based Deployment with Cloud Storage This deployment is similar to DM5 in that the access logic is deployed in the edge layer. However, in DM5 attributes are stored and retrieved from a remote repository deployed in the edge node. When an access request is received at the edge layer, the access control mechanism retrieves the attributed needed for policy evaluation from the remote cloud-based repository and uses those attributes for the evaluation of the access request.

DM7: Edge-based Deployment with Local Storage This deployment is similar to DM6 in that the access logic is deployed in the edge layer. However, in DM7 attributes are stored and retrieved from a local repository deployed in the edge node. When an access request is received at the edge layer, the access control mechanism retrieves the attributed needed for policy evaluation from the local repository and uses those attributes for the evaluation of the access request.

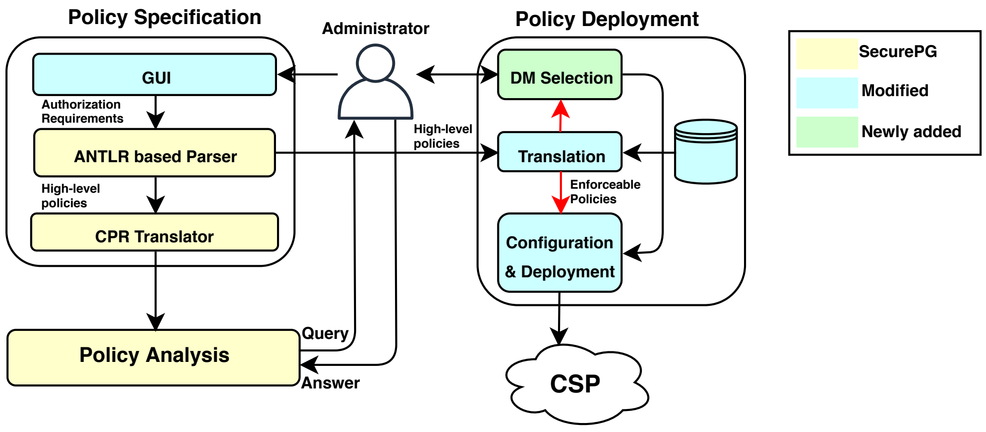


Figure 7.2: Architecture of SECUREPG extension for IoT

7.4 Tool-supported Policy Configuration

While the enhanced capabilities of the access control mechanism provided by deployment models DM2/DM3 and DM5/DM6/DM7 allow a more fine-grained control of IoT resources, it introduces additional burden in the configuration and deployment of the infrastructure needed for policy evaluation. To this end, we have extended SECUREPG [MR17], a policy authoring framework for cloud environment, to support the configuration and deployment of the infrastructure necessary to support policy evaluation and enforcement in IoT platforms. In this section, we discuss the extension of SECUREPG for IoT.

7.4.1 SECUREPG Extension for IoT

To support the configuration of the access control mechanism in the possible deployment models of cloud-edge IoT, we have extended SECUREPG to enable the configuration of an arbitrary number of IoT entities with an unbounded number of attributes and their deployment in AWS IoT and Greengrass. Figure 7.2 shows the modified components in light blue color and newly added components in light green color. Figure 7.3 shows the configuration and deployment procedure supported by the extended SECUREPG where red dashed rectangle denotes the extended functionalities.

Entity and Policy Configuration 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 7.1 presents the concepts that have been integrated into SECUREPG, namely a representation of the physical devices, called *clients*, and their virtual counterpart (in the cloud), called *things*. Things are organized in groups and possess a specific set

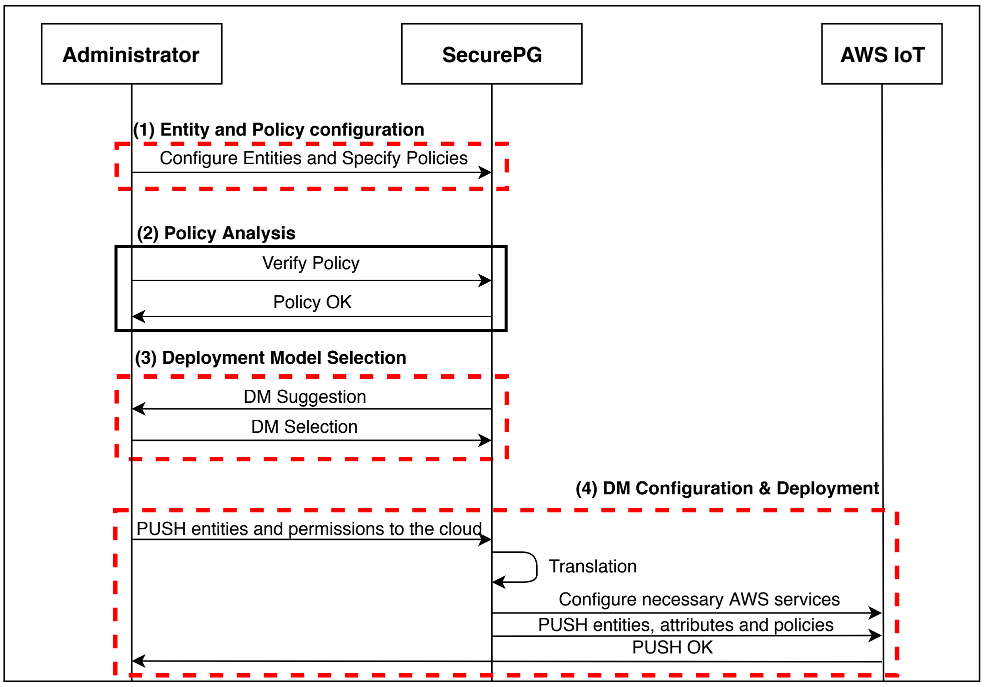

Administrator
SecurePG
AWS IoT
(1) Entity and Policy configuration
Configure Entities and Specify Policies
(2) Policy Analysis
Verify Policy
Policy OK
(3) Deployment Model Selection
DM Suggestion
DM Selection
(4) DM Configuration & Deployment
PUSH entities and permissions to the cloud
Translation
Configure necessary AWS services
PUSH entities, attributes and policies
PUSH OK

Figure 7.3: SECUREPG Configuration and Deployment Procedure for IoT Platforms

of attributes. IoT resources are represented in terms of *topics* and *topic filters*. As in AWS, the types of subjects and resources specified by policy administrators in their policies are bound to specific actions.

Table 7.1: IoT Entities and Actions supported by SECUREPG

Entities		Actions	Description
IoT Subject	Client	Connect	Support the IoT physical devices connection.
	Things, Things 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.

Policy Analysis The specified policies are analyzed and validated using the module already available in SECUREPG, which analyzes the defined policies and reports possible policy misconfigurations before policies are deployed.

Deployment Model (DM) Selection We have extended SECUREPG with a new component that assists users in the selection of the deployment model. This component analyzes the defined access control policies and, based on the granularity of permissions and the choice of the IoT platform (i.e., cloud-based or edge-based), the tool suggests a possible deployment model. For instance, in case the user wants to use a cloud-based solution, if the policies contain three or less attributes (in Condition),[1] the tool suggests DM1 in case of pure cloud architecture. Otherwise, if policies contain more than three attributes, the tool suggests using DM2 or DM3.

Deployment Model Configuration & Deployment We have extended the component provided by SECUREPG to support the configuration and deployment of policies and the necessary infrastructure with respect to the selected deployment model. If a cloud-based deployment model (DM1, DM2 or DM3) is selected, the tool configures the cloud-based IoT platform by triggering the deployment of configurations on AWS IoT. In case DM1 is selected, all entities are configured using AWS IoT native mechanisms. On the other hand, if DM2 or DM3 is selected, this component creates and configures the Lambda function infrastructure necessary to support policy evaluation; this requires configuring a Custom Authorizer that coordinates policy evaluation.

When an edge-based deployment model (DM4, DM5, DM6 or DM7) is selected, the component configures the Greengrass environment in AWS IoT, which in turn deploys the configurations (e.g., entities, topics) on a Greengrass core. In case DM4 is selected, all entities are configured using the AWS Greengrass' native mechanism. In case of DM5, DM6 or DM7, the component also creates and deploys the Lambda function infrastructure to support policy evaluation on the Greengrass core along with, if necessary, a local database for the storage of attributes and policies.

IoT entities' configurations and policies are also deployed in the IoT platform. In particular, the tool 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.

7.5 A prototype of the smart lock scenario

In this section, we present our realization of the smart lock scenario on AWS IoT and Greengrass using the deployment models presented in the previous section.

7.5.1 Smart Lock System

We have realized a smart lock system following the possible deployment models for cloud-edge IoT solutions. The system comprises a smart lock that users can open and close using an application installed on their mobile device. The employed smart lock lacks direct connectivity to the Internet and, thus, it relies on the user's mobile device for Internet connectivity.

[1]Recall that AWS IoT supports the specification of up to three attributes in policies.

```
smart_lock:
    platform: mqtt
    name: frontdoor
    state_topic: "home/frontdoor/"
    command_topic: "home/frontdoor/set"
    payload_lock: "LOCK"
    payload_unlock: "UNLOCK"
    ...
```

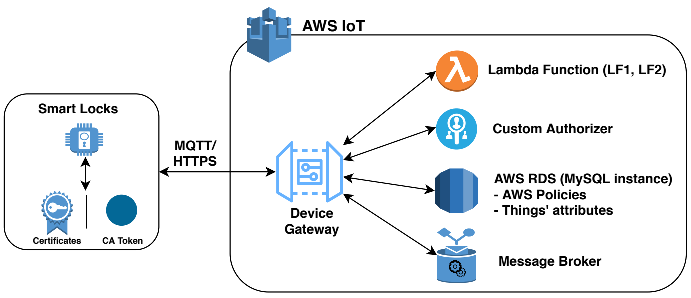
Figure 7.4: Message broker configuration

Access to the smart lock is regulated by a remote authorization mechanism. The interaction between the smart lock and the remote authorization mechanism is managed by an IoT endpoint via a MQTT message broker. The message broker uses *topics* to route messages from the publishing entities (i.e., the mobile device) to the subscribed entities (i.e., the smart lock). The subscription of IoT subjects to IoT resources is defined using *topic filters*. Figure 7.4 shows the configuration of the MQTT-based smart lock installed on the front door of a home. The smart lock has a *state_topic* to publish state changes. The user can change the state of the lock (i.e., *LOCK/ UNLOCK*) by publishing on *command_topic*.

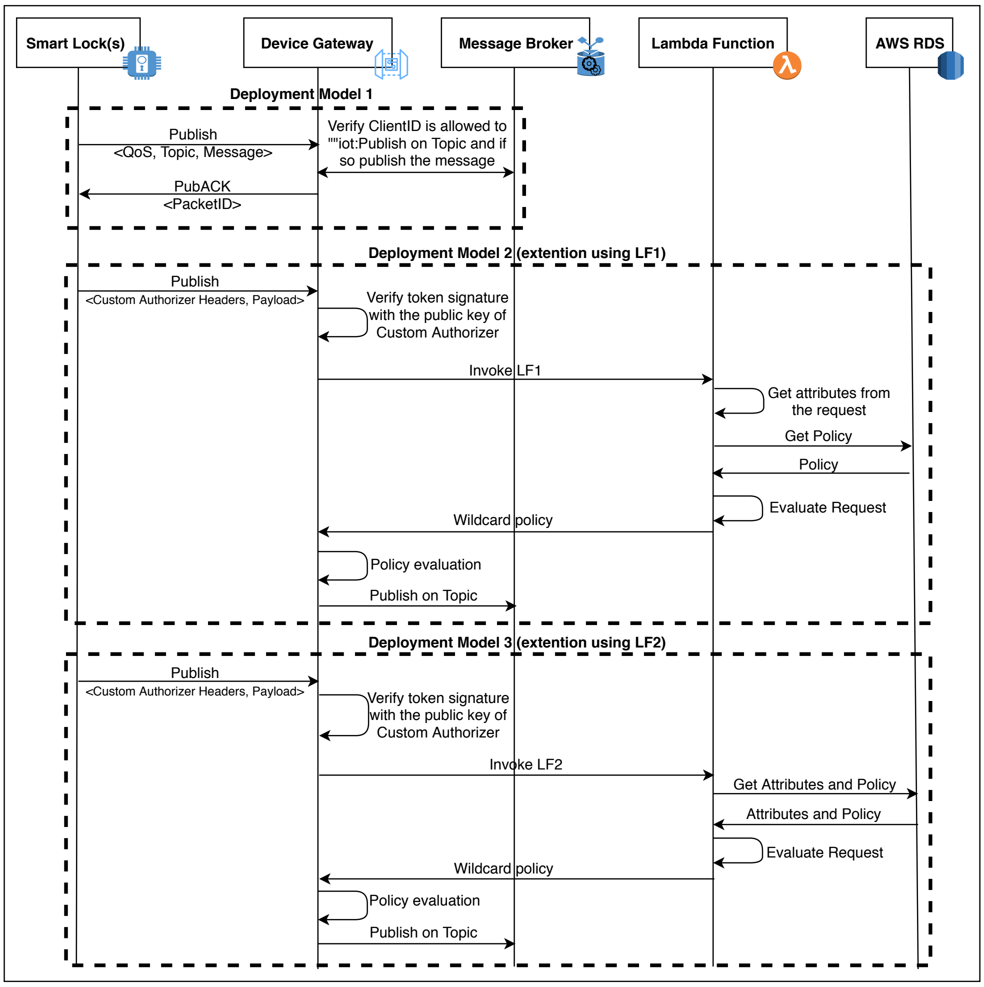
Cloud-based Deployment We configured AWS IoT to realize the smart lock system according to the cloud-based architectures. As shown in Figure 7.5a, AWS IoT relies on the Device Gateway to enables the interaction between cloud-based applications and the smart lock, thus acting as the IoT endpoint. Moreover, our smart lock system uses several services offered by AWS IoT such as Lambda functions, message broker, AWS RDS and Custom Authorizer.

When connecting to the Device Gateway in AWS IoT, the smart lock has to provide its ID and X.509 certificate. These are used by the Device Gateway to authenticate the smart lock. Once connected (Figure 7.5b), the smart lock can publish messages on any authorized topic. Additionally for DM2 and DM3, we extended the access control mechanism of AWS IoT by configuring a *Custom Authorizer* that issues authentication tokens to clients. In order to connect to the Device Gateway using the Custom Authorizer, the smart lock should provide a valid token issued by the Custom Authorizer. The difference between DM2 and DM3 lies in the approach used to retrieve the attributes needed for policy evaluation (Figure 7.5b). In DM2, the attributes are retrieved from access requests and policies from AWS RDS, whereas, in DM3, both the attributes and policies are fetched from a MySQL instance of AWS RDS (i.e. user-defined external storage).

Edge-based Deployment To deploy the access control logic in the edge (DM4, DM5, DM6 and DM7), we configured a number of AWS Greengrass Cores on Raspberry Pi 3 Model B boards. As shown in Figure 7.6a, the smart lock is configured to interact with Greengrass core on a local network. The Greengrass core enables the local execution of Lambda functions, messaging (via the message broker) and security (via subscription lists). In addition, it allows interaction with



(a) Prototype Components


Smart Lock(s)
Device Gateway
Message Broker
Lambda Function
AWS RDS
Deployment Model 1
Publish
<QoS, Topic, Message>
Verify ClientID is allowed to ""iot:Publish on Topic and if so publish the message
PubACK
<PacketID>
Deployment Model 2 (extention using LF1)
Publish
<Custom Authorizer Headers, Payload>
Verify token signature with the public key of Custom Authorizer
Invoke LF1
Get attributes from the request
Get Policy
Policy
Evaluate Request
Wildcard policy
Policy evaluation
Publish on Topic
Deployment Model 3 (extention using LF2)
Publish
<Custom Authorizer Headers, Payload>
Verify token signature with the public key of Custom Authorizer
Invoke LF2
Get Attributes and Policy
Attributes and Policy
Evaluate Request
Wildcard policy
Policy evaluation
Publish on Topic

(b) Workflow

Figure 7.5: Cloud-based Architecture

a remote AWS storage service, i.e. AWS RDS, or a local MySQL database deployed on the Greengrass core.

To connect to the IoT endpoint (i.e., the Greengrass core), the smart lock has to provide its ID and X.509 certificate configured with a reference to the Greengrass core, the sender and receiver IoT things, and subscription rules. The Greengrass core authenticates the smart lock through the certificate and authorizes it according to the subscription rules specified in the subscription table. Subscription rules specify the topic on which a client can send messages and which clients are subscribed to the topic.

In DM4, subscription rules have the form $\langle sender, MQTT\_topic, IoT\ Cloud\rangle$ and $\langle IoT\ Cloud, MQTT\_topic, receiver\rangle$ representing the permission to send and receive a message respectively.

On the other hand, DM5, DM6 and DM7 use subscription rules of the form $\langle sender, MQTT\_topic, Lambda\_function\rangle$ and $\langle Lambda\_function, MQTT\_topic, receiver\rangle$, where *Lambda_function* is a reference to the Lambda function used to make access decision (we refer to the next section for a description of the implemented Lambda functions).

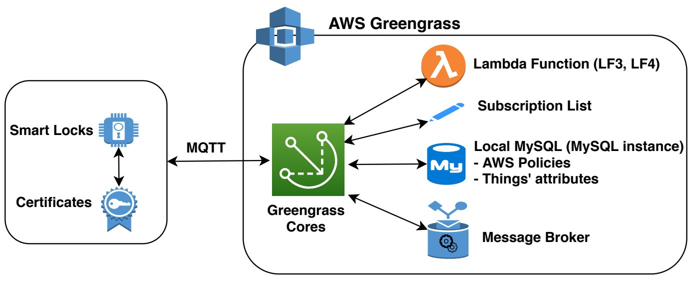
We implemented three variants of DM4 that differ for the way in which the attributes and policies needed for policy evaluation are retrieved (Figure 7.6b). In one variant (i.e. DM5), the attributes are retrieved from the access request and policies are fetched from the local mySQL database deployed in the Greengrass core. In the other two variants (DM6 and DM7), the attributes and policies are respectively fetched from the remote cloud-based storage i.e. AWS RDS and local MySQL database deployed on the Greengrass core.

7.5.2 Lambda Functions

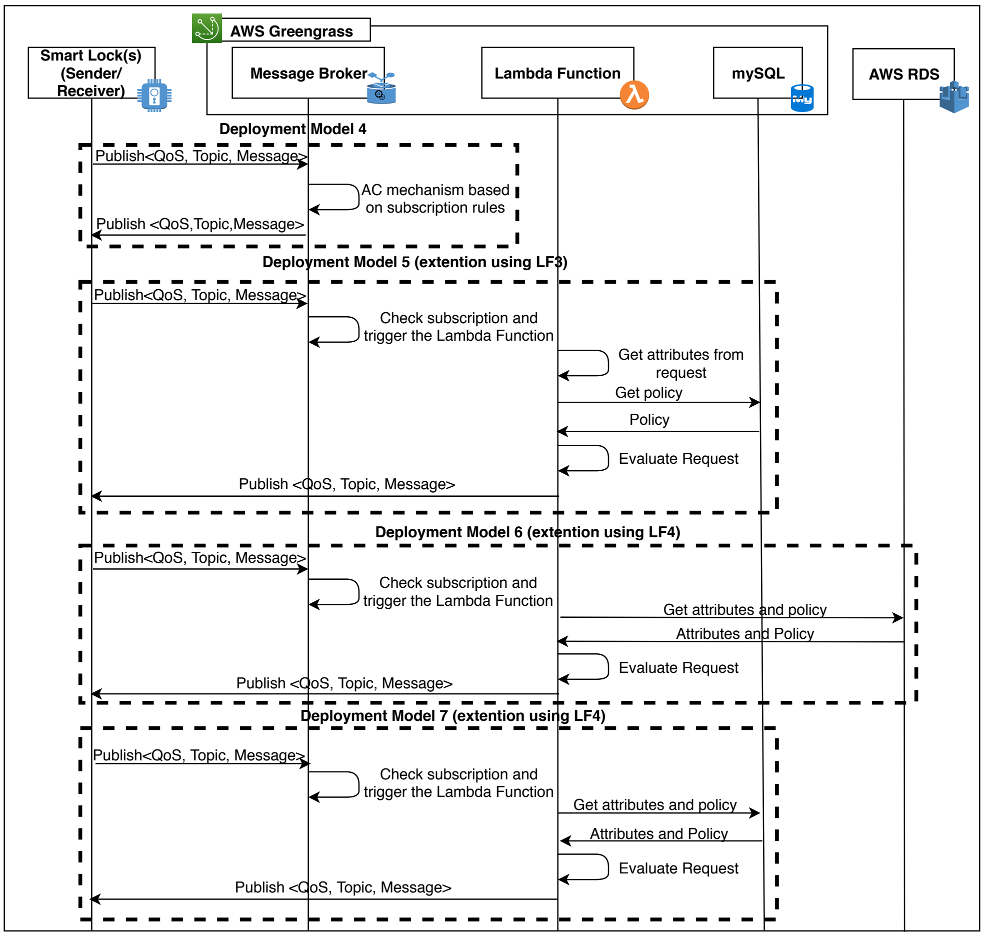
To extend the capabilities of pure cloud solutions in order to support the evaluation of policies with an arbitrary number of attributes (DM2 and DM3) and to enable attribute-based access control in AWS Greengrass (DM5, DM6 and DM7), we have implemented four Lambda functions, two for AWS IoT (referred to as LF1 and LF2 in Figure 7.5b) and two for AWS Greengrass (referred to as LF3 and LF4 in Figure 7.6b).

LF1 extends the capabilities of the native authorization mechanism of AWS IoT (DM1). The use of this function requires configuring a *Custom Authorizer* that uses the attributes in the request to configure the Lambda function and then to invoke it in order to determine whether the client is allowed to publish and/or subscribe to a certain topic. It is worth noting that the size of the request is fix and only allows the specification of a limited number of attributes (up to 30 in this case from the smart home use case [HGP+18]).

LF2 extends LF1 by addressing the limitation on the number of attributes that can be used



(a) Prototype Components


AWS Greengrass
Smart Lock(s) (Sender/ Receiver)
Message Broker
Lambda Function
mySQL
AWS RDS
Deployment Model 4
Publish<QoS, Topic, Message>
AC mechanism based on subscription rules
Publish <QoS,Topic,Message>
Deployment Model 5 (extention using LF3)
Publish<QoS, Topic, Message>
Check subscription and trigger the Lambda Function
Get attributes from request
Get policy
Policy
Evaluate Request
Publish <QoS, Topic, Message>
Deployment Model 6 (extention using LF4)
Publish<QoS, Topic, Message>
Check subscription and trigger the Lambda Function
Get attributes and policy
Attributes and Policy
Evaluate Request
Publish <QoS, Topic, Message>
Deployment Model 7 (extention using LF4)
Publish<QoS, Topic, Message>
Check subscription and trigger the Lambda Function
Get attributes and policy
Attributes and Policy
Evaluate Request
Publish <QoS, Topic, Message>

(b) Workflow

Figure 7.6: Edge-based Architecture

for policy evaluation. In particular, this function can be used to retrieve an arbitrary number of attributes stored in a MySQL instance of AWS RDS. The MySQL instance comprises two tables, one storing JSON policies (specified using the AWS syntax) and one storing attributes characterizing AWS things.

LF3 extends the capabilities of the authorization mechanism provided by AWS Greengrass (DM4). The functionality of LF3 is similar to LF1 but it is deployed on the Greengrass core. Similar to LF1, it fetches attributes from the token inside the message field of the Publish request. However the approach is restricted with a limit on the maximum number of specified attributes due to the fixed header length of Custom Authorizer.

LF4 extends LF3 to address the bottleneck caused by the retrieval of attributes from the access request. In particular, LF4 fetches attributes and policies from a remote storage deployed in the back-end cloud (DM6) or local storage on the Greengrass core (DM7).

Implementation Lambda functions are configured by accessing the Lambda Service on the AWS management console. The Lambda functions are implemented in Java8 and uploaded to the AWS Lambda service.[2]

The Lambda management interface allows the specification of several *Environment Variables* as key value pairs that are accessible by the Lambda functions.The *Environment Variables* are useful to store configuration settings without the need to change function code and are specified on the basis of the deployment model. For example, they are used to configure the connection with AWS RDS in case of DM3 and with the MySQL database deployed on the Greengrass core in case of DM6.

[2]The code is available as a Maven project at `https://goo.gl/xybfGf`.

Chapter 8

ACaaS Evaluation

In this chapter, we would like to assess to what extent the deployment models presented in Chapter 7 meet the requirements identified in Chapter 3. To this end we perform an extensive experimental evaluation of these deployment models with respect to the latency (AC5), reliability (AC6) and scalability (AC7) using our prototype implementation. We do so using our prototype implementation of the smart lock system in different configurations – integrating cloud and edge computing together with the defined Lambda functions (c.f. Section 7.5.2).

In this chapter we are particularly interested in answering the following research questions:

RQ1 Does the use of edge computing affect the performance of the authorization mechanism?

RQ2 Does the use of Lambda functions for access control influence the performance of the IoT deployment?

RQ3 Does the approach adopted for attribute storage and retrieval affect the performance of the authorization mechanism?

The use of edge computing provides substantial benefits (in terms of latency, reliability and scalability) compared to cloud-based solutions [Bye17]. However, the impact of edge-based solutions on the authorization mechanism is still unclear. The first research question (RQ1) aims to investigate this aspect through a comparative analysis between cloud-based deployment models (DM1, DM2 and DM3) and edge-based deployment models (DM4, DM5, DM6 and DM7) of the smart lock deployment.

We have extended the capabilities of the native access control mechanism provided by AWS IoT and Greengrass by exploiting the extensibility point (i.e., Lambda functions) provided by AWS IoT and Greengrass. While achieving a high level of expressibility [AMRZ18a], the use of Lambda functions for access control may impact the performance of the IoT ecosystem. The second research question (RQ2) aims to quantify this impact through a comparative analysis between the deployment models employing the AWS native access control mechanism (DM1 and DM4) and deployment models employing the enhanced access control mechanism (DM2/DM3 and DM5/DM6/DM7).

Different approaches can be used to retrieve the attributes needed for policy evaluation, each approach imposes different requirements on the underlying hardware and software. It is important to balance the cost of hardware and software with efficiency based on the requirements of the specific use case of IoT [HS12]. The third research question (RQ3) aims to investigate this aspect through a comparative analysis of the mechanisms for attribute retrieval with respect to both architectural approaches (i.e., DM1, DM2 and DM3 for cloud-based architectures and DM4, DM5, DM6 and DM7 for edge-based architectures).

8.1 Settings

To answer the questions above we have performed two sets of experiments. The first set aims to assess processing time and failure rate, whereas the second set aims to assess the throughput for each deployment model. Next, we first present the general configuration of the experiments and, then we present the configuration specific to each experiment.

8.1.1 Experiment setup – General

We performed our experiments using AWS IoT Greengrass Core Version 1.9.0 and a free-tier AWS account deployed in the Europe (Frankfurt) eu-central-1 region.[1] To evaluate the performance, we used JMeter with MQTT JMeter Plugin [EMQ17], which extends JMeter's capability to test the functional behavior and measure performance against the MQTT protocol.

To evaluate the deployment models, we configured a fleet of policy-enabled IoT things in AWS IoT, all belong to a single thing type named Lock and associated with an AWS IoT certificate. Table 8.1 shows the configuration used for performance evaluation of each deployment model.

The experiments mainly relied on MQTT protocol with X.509 certificates on port 8883; however, in case of DM2 and DM3, HTTPS with Custom Authorizer tokens on port 443 are used to invoke Lambda functions (i.e., LF1 and LF2). Therefore, in case of DM2 and DM3, an HTTPS Post request is used to Publish a message on the specified Topic, whereas in all other cases MQTT Publish request is used to Publish a message on the specified Topic.

In both cases, we record the time a request to publish a message is sent and the time the request is processed. The difference between these timestamps is used to determine the communication

[1]AWS Greengrass service is not available in every region.

	Protocol	Identity Type	Port	Message Type	TimeStamp 1	TimeStamp 2
DM1	MQTT	X509 Certificate	8883	Publish	Smart Lock (Publish Message Send)	When PubACK is received by the sender
DM2 (LF1)	HTTPS	Custom Authorizer Token	443	POST-Publish	Smart Lock (Publish Message Send)	Smart Lock (when HTTP code 2xx is received)
DM3 (LF2)	HTTPS	Custom Authorizer Token	443	POST-Publish	Smart Lock (Publish Message Send)	Smart Lock (when HTTP code 2xx is received)
DM4	MQTT	X509 Certificate	8883	Publish	Smart Lock (Publish Message Send)	Message received by the subscribed Clients
DM5 (LF3)	MQTT	X509 Certificate	8883	Publish	Smart Lock (Publish Message Send)	Greengrass Group (after LF publish the authorized message on the requested topic at message broker)
DM6 (LF4)	MQTT	X509 Certificate	8883	Publish	Smart Lock (Publish Message Send)	Greengrass Group (after LF publish the authorized message on the requested topic at message broker)
DM7 (LF4)	MQTT	X509 Certificate	8883	Publish	Smart Lock (Publish Message Send)	Greengrass Group (after LF publish the authorized message on the requested topic at message broker)

Table 8.1: Experimental Configurations

and processing time for the evaluation of a IoT client's request in AWS IoT and Greengrass. Note that the difference in message protocol (MQTT and HTTPS) and involvement of additional component at the edge level (i.e., Greengrass Core) used for deployment of cloud and edge-based architectures do not allow to maintain a fixed criterion for time stamping. We discuss this point further at the end of the section.

We configured a total of 100 sender clients that connects with their AWS certificate and private key on the AWS IoT endpoint (using port 8883), publish and disconnect. The clients are configured with 30 attributes. A total of five policies are created by varying the number of attributes in the condition tag (i.e., 0, 5, 10, 20 and 30 attributes). The policies are associated to the certificate of the specific client, one at a time, depending on the test. In DM2 and DM3, the connecting clients publish with the Custom Authorizer headers on the AWS IoT endpoint (using HTTP on port 443). The Custom Authorizer is triggered only if the signature of the token is made with the Custom Authorizer private key and connects to RDS. All the components that support the custom authorizer are configured in the AWS IoT, RDS and Lambda services.

Since Greengrass uses a rudimentary access control mechanism based on subscriptions (cf. Section 2.4.2), we did not consider attributes for DM4. On the other hand, Lambda functions LF3 and LF4 allow the retrieval of a potentially unlimited number of attributes from MQTT messages. Thus, we evaluated DM5 to DM7 up to 30 attributes to compare with cloud-based deployments.To test deployment models DM5 to DM7, a total of 200 clients were configured to connect and publish messages on the subscribed topics, where 100 acting as Publisher clients and 100 acting

	No. of Devices	No. of Edge Nodes	No. of Attributes
DM1	1,10,50,100	NA	0,5,10,20,30
DM2	1,10,50,100	NA	0,5,10,20,30
DM3	1,10,50,100	NA	0,5,10,20,30
DM4	1,10,50,100	1,5,8*	0
DM5	1,10,50,100	1,5,8*	0,5,10,20,30
DM6	1,10,50,100	1,5,8*	0,5,10,20,30
DM7	1,10,50,100	1,5,8*	0,5,10,20,30

* When 1 or 10 devices are used, we only consider 1 edge device. In case of 50, and 100 devices, we test the deployment model using 1, 5 and 8 edge devices.

Table 8.2: Experimental Setup - Single Request

as Subscriber clients.

8.1.2 Experimental Setup – Single Request

To measure processing time and failure rate, we evaluated the deployment models against single request, where each connected device sends a single publish request at a time. The configuration details are given in Table 8.2. The processing time and failure rate are computed for each deployment model by varying the number of devices and policy size (i.e., the number of attributes in the condition tag) with the exception of DM4. As seen in Table 8.2, eight Greengroups (one on each Raspberry PI) are configured with subscriptions in the form <Client_[sender], TestingDM4/#, Client_[receiver]>. The publish requests in edge-based deployments (i.e., DM4-DM7) are evaluated by varying the number of edge nodes (i.e., 1, 5 and 8).

8.1.3 Experimental Setup – Parallel Request

To measure the throughput and failure rate of the deployment models, we evaluated the deployment models against parallel request, where all connected devices send publish request continuously. The configuration details are given in Table 8.3. The main configuration stays the same as for single request test. However, by using a JMeter concurrency Thread Group, each connected client in step size of 10 (from device 1 to 100) send a Publish request in parallel. A new batch of 10 clients is added each minute. The experiments are performed with 30 attributes except for DM4, they are performed with zero attributes due to the limitations of the access control mechanism provided by Greengrass (cf. Section 2.4.2). In the evaluation of edge-based deployments, the publish request are sequentially distributed on the available edge nodes (1-8) while maintaining a maximum of 50 request per edge device.

8.1.4 Limitations

The deployment of the smart lock system in AWS required different configurations depending on the deployment model (cf. Table 8.1). We now discuss how these differences can affect our experiments and how we addressed them.

In all deployments, processing time was determined based on the time a message request is send and the time a message response is received by the client. However, different architectural choices required different approaches for recording these timestamps. As AWS IoT natively uses MQTT with acknowledgement, for DM1 we recorded the time an MQTT publish request is sent by the client and the time the MQTT acknowledge message (i.e., PubACK) is received by the client. Similarly, in DM2 and DM3 we recorded the time the smart lock sends a HTTPS POST request and the time the corresponding HTTP acknowledge is received by the smart lock. On the other hand, Greengrass does not support acknowledgements for messages routed locally at the edge node. To obtain results comparable to cloud-based deployments, we generated response messages that mimic the acknowledgement messages used in cloud-based deployments. Accordingly, we record the time the smart lock sends an MQTT publish request and the time the acknowledgment message is received by the smart lock. It is worth noting that both the times in which the publish request is sent and the acknowledgement is received are recorded on the client side and, therefore, the computed processing time is comparable across the different deployments.

The differences in the messaging protocols used to realize the deployment models (HTTPS for DM2 and DM3 and MQTT for the other deployment models) result in different request-response patterns. MQTT is a asynchronous message protocol based on one-to-many relationship whereas HTTPS is a synchronous message protocol based on one-to-one relationship.[2] To obtain comparable results across the deployment models, we configured MQTT to behave as a one-to-one relationship.

In the second experiment (parallel requests), we were unable to reliably measure failure rate for edge-based deployments. In DM4, no information on failed request is available as Greengrass does not log messages routed locally or acknowledge them; therefore, we can not argue that all published messages should have arrived at destination. In case of the edge-based deployments using Lambda functions (DM5, DM6, DM7), an analysis of the logs recorded by the Lambda infrastructure shows a failure rate of zero percent. This highlights that all messages intercepted by Lambda functions are successfully processed. However, this approach does not allow tracing requests that are not processed by the Lambda function. To this end, we did not consider failure

[2]https://www.hivemq.com/blog/mqtt5-essentials-part9-request-response-pattern

	No. of Devices	No. of Edge Nodes	No. of Attributes
DM1	1-100	NA	30
DM2	1-100	NA	30
DM3	1-100	NA	30
DM4	1-100	1-8*	0
DM5	1-100	1-8*	30
DM6	1-100	1-8*	30
DM7	1-100	1-8*	30

* The request are sequentially distributed with a maximum of 50 request per edge device.

Table 8.3: Experimental Setup - Parallel Request

rate for edge-based deployments in the discussion of the results.

8.2 Evaluation Metrics

We evaluated the results of our experiments using three evaluation metrics [BA13], namely *processing time*, *failure rate*, and *throughput*. Table 8.4 shows how these metrics are related to the latency, reliability and scalability requirements.

Table 8.4: Evaluation Metrics

Requirements	Evaluation Metrics		
	Processing Time	Failure Rate	Throughput
Latency	×	—	—
Reliability	×	×	—
Scalability	—	—	×

Processing Time This metric measures the time needed to process a request. It can be expressed as:

$$Processing\ Time = T_{Processed} - T_{Request} \tag{8.1}$$

where $T_{Request}$ is the time in which a request is send by the client, and $T_{Processed}$ is the time in which the message broker completes the processing of the request.

Failure Rate This metric assesses the ability of the message broker to correctly process requests within a maximum acceptable time. It is measured as the percentage of failed message requests over the total number of message requests attempted. It can be formally expressed as:

$$Failure\ Rate = \frac{Message\ Requests\ Failed}{Message\ Requests\ Attempted} \tag{8.2}$$

where *Message Requests Failed* represents the number of requests that the message broker fails to process within the maximum acceptable time and *Message Requests Attempted* represents the total number of requests sent to the message broker.

Throughput This metric measures the number of requests successfully processed per time unit. It can be formally expressed as:

$$Throughput = \frac{Message\ Request\ Processed}{Time} \tag{8.3}$$

where *Message Request Processed* denotes the number of requests successfully processed by the message broker and *Time* represents the time unit.

8.3 Results

We now present the results of our experimental evaluation of the deployment models presented in Chapter 7 with respect to processing time, failure rate and throughput using our prototype implementation.

8.3.1 Processing Times

Figure 8.1 shows the comparison of processing time between the native cloud-based solution (DM1) and the native edge-based solution (DM4) in various configurations (i.e., 1, 5 and 8 edge nodes). Although edge-based solutions are expected to outperform cloud-based solutions, we found that DM1 only takes 21ms on average to Publish a message on a specified topic whereas DM4 takes 50 ms. We further investigated this issue by repeating the experiments for DM1 in a different AWS region (i.e., US Oregon Region). Figure 8.2 shows the performances of AWS significantly differ from region to region. Figure 8.1 also shows that the processing time of the edge-based deployment (DM4) decreases when the publish requests are distributed on multiple edge nodes (i.e., 5 and 8 Raspberry PIs for 50 or more device requests) compared to the use of a single edge node.

Figure 8.3 shows the processing time for cloud-based deployment models (DM1, DM2 and DM3) with respect to the number of devices. We can observe that the performance of DM1 is stable when the number of devices increases. In contrast, a high variation in processing time is observed for the cloud-based deployment models involving Lambda functions (i.e., DM2 and

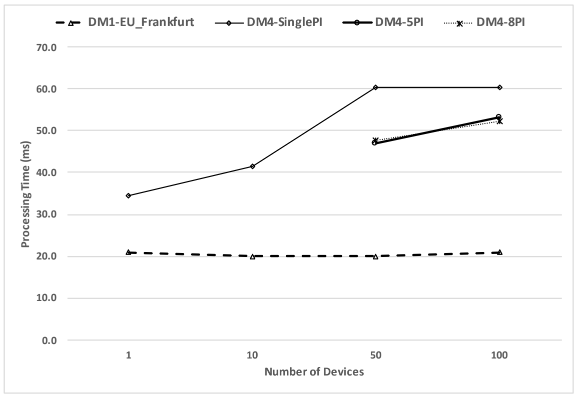


Figure 8.1: Processing time (in ms) for AWS IoT (DM1) and Greengrass (DM4) access control mechanisms

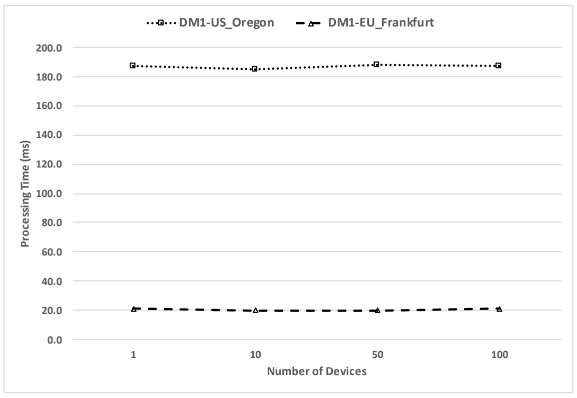


Figure 8.2: Processing time (in ms) for cloud-based deployment model (DM1) w.r.t. AWS IoT regions

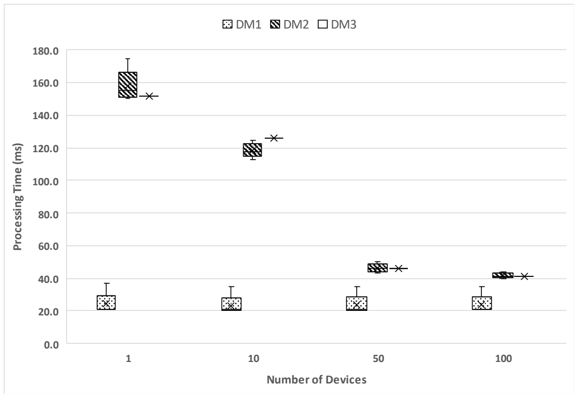


Figure 8.3: Processing time (in ms) for the cloud-based deployment models w.r.t. the number of devices

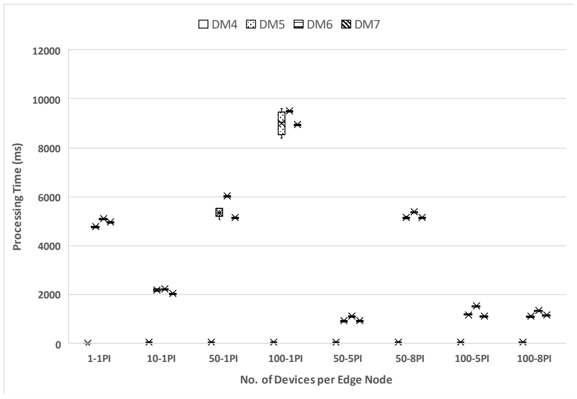


Figure 8.4: Processing time (in ms) for the edge-based deployment models w.r.t. the number of devices

DM3). The improvement in performance mainly due to how AWS IoT manages resources. The first time a Lambda function is invoked, it connects to other services (such as storage) and loads the necessary components (such as attributes) in the memory. For subsequent invocations of the Lambda function, it relies on the pre-fetched components in the memory.

Figure 8.4 shows the processing time for edge-based deployments (DM4, DM5, DM6 and DM7). We can observe that DM4, which uses the subscription-based mechanism provided by AWS Greengrass, outperforms edge-based deployment models that employ Lambda functions to enable attribute based access control (DM5, DM6 and DM7). The figure also shows the positive impact of using multiple edge nodes.

We also investigated the processing time of the deployment models with respect to policy size.

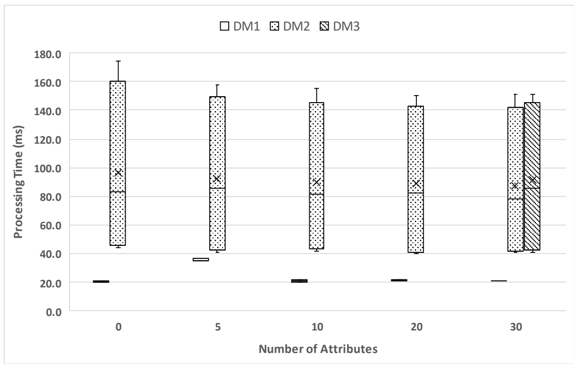


Figure 8.5: Processing time (in ms) for the cloud-based deployment models w.r.t. number of attributes

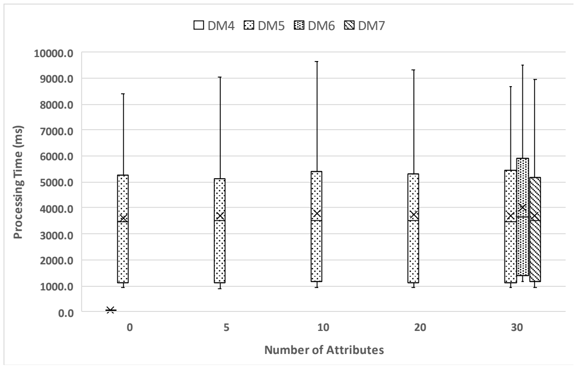


Figure 8.6: Processing time (in ms) for the edge-based deployment models w.r.t. number of attributes

The results are reported in Figure 8.5 for the cloud-based deployment models (DM1, DM2 and DM3) and in Figure 8.6 for the edge-based deployment models (DM4, DM5, DM6 and DM7). We can observe that DM1 is more efficient compared to the cloud-based deployments using Lambda functions (i.e., DM2 and DM3). However, there is not a significant difference among DM2 and DM3, thus highlight that the methods in which attributes are provided to the policy decision point (i.e., either provided in access request or fetched from AWS RDS) does not effect the performance of the access control mechanism. On the other hand, from Figure 8.6 we can observe that DM4 outperforms the other deployment models that enable ABAC using Lambda functions. Moreover, among those deployment models, the ones based on a local mySQL database (i.e., DM5 and DM7) for storage and retrieval of attributes and policies perform slightly better than DM6, which retrieves attributes and policies from AWS RDS.

8.3.2 Throughput

Figure 8.7 shows the results concerning throughput for DM1 and DM4. We can observe a steady increase in throughput for DM1 with the increase in number of devices, supporting up to 2685 requests processed per second. We speculate that the high throughput in DM1 is due to the scalable infrastructure offered by AWS. On the other hand, DM4 initially provides a very high throughput (7122 requests per second), which reduces gradually with the increase in number of devices that are continuously sending messages in parallel. However, due to limited number of resources at the edge layer, once those resources are saturated the throughput stabilizes around 2000 requests per second.

Figure 8.8 shows the throughput of deployment models using Lambda functions (DM2, DM3, DM5, DM6 and DM7). We can see a gradual increase in throughput for cloud-based deployments (DM2 and DM3), whereas a gradual decrease in throughput can be seen for edge-based deploy-

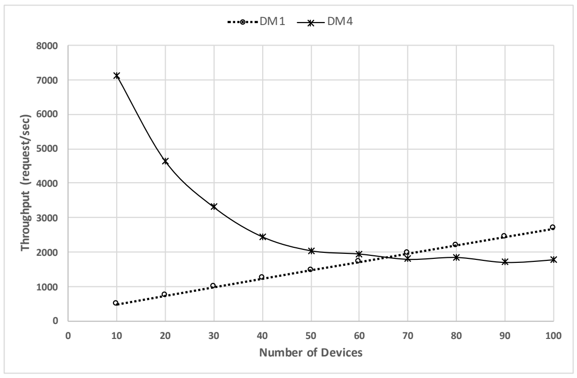


Figure 8.7: Throughput for AWS IoT (DM1) and Greengrass (DM4) access control mechanisms

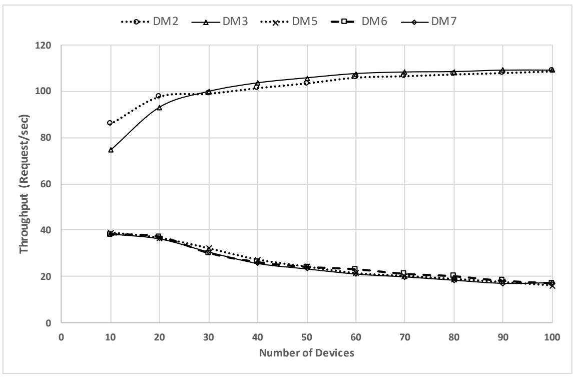


Figure 8.8: Throughput for cloud-based and edge-based deployment models using Lambda functions

ments (DM5, DM6, DM7). Factors such as the functioning and management of Lambda functions contribute to the poor throughput of these deployment models as compared to their respective cloud and edge counterparts (i.e., AWS IoT and Greengrass respectively) as shown in Figure 8.7. Moreover, the constrained nature of edge devices also contribute to the poor performance of Lambda functions deployed in those devices.

8.3.3 Failure Rate

The experiments using single requests shows a failure rate of zero percent for all deployment models. However, as seen in Figures 8.3 and 8.4, this is achieved at the cost of higher processing time for the deployment models using Lambda functions (DM2, DM3, DM5, DM6 and DM7) compare to their cloud/edge native counterparts (i.e., DM1 and DM4 respectively).

Similarly, a failure rate of zero percent is noted in the test for parallel requests for cloud-based deployments (DM1, DM2, DM3). However, as discussed in the limitations, we do not have an adequate mechanism to measure the exact failure rate for the edge-based deployments.

8.4 Discussion

We now discuss the results of our experiments with respect to the research questions. Two remarks are in order. First, Due to the limited insights that can be inferred from the results concerning failure rate, therefore, we are only considering processing time and throughput in our discussion. Second, AWS IoT and Greengrass support extensibility point that are used in this work for the realization of access control mechanism using lambda function, therefore, (AC4) requirement is satisfied at both the cloud and edge level.

RQ1. This question aims to assess the impact of edge computing on the performance of the authorization mechanism. Our experiments show that the processing time taken by the edge based deployment (DM4) with multiple edge nodes is twice the processing time taken by the cloud-based deployment (DM1). This can be attributed to the constrained nature of Greengrass (i.e., software component) and the Raspberry PI (i.e., hardware component). For instance, within Greengrass, we experienced an exponential growth in the processing time when a single device was deployed, however, that was normalized when more Greengrass cores were deployed.

The pure cloud and edge-based solution (i.e., DM1 and DM4 respectively) strongly depends on the cloud services offered by the Cloud Service Provider (CSP) and its access control capabilities. In the following, we perform the comparison of these deployment model using our experience with AWS IoT platform in the context of access control for IoT.

- (AC1) Recalling the limitation of AWS IoT & Greengrass as discussed in Section 2.4.3, we can argue that DM1 fails to fully support expressiveness whereas the rudimentary access control provided by AWS at the edge level fails to support this requirement. For instance in DM1, when handling a fleet of smart locks and user devices, system administrator would require more than three subject's attributes and most importantly, resource's attributes to configure the polices. Similarly, Greengrass uses a simple subscription-based authorization mechanism in which access rights are granted only considering the device's identifiers.
- (AC2) Both AWS IoT (DM1) and Greengrass (DM4) provide user with very little, almost no support for policy administration.
- (AC3) The use of a proprietary, ad-hoc access control mechanism in both DM1 and DM4 prevent the satisfaction of (AC3) and limit the portability of access control policies across various IoT platform.
- (AC5) The performance of DM1 suffers from the propagation delay associated with the offloading of access control to the cloud, whereas, the resources in edge computing are strategically placed near the users do not incur any propagation delays. This lead to the partial satisfaction of latency in case of DM1 as compare to the complete satisfaction in DM4.
- (AC6) DM1 is not resilient in case of connection problems between users' devices and the cloud whereas the resources in DM4 being strategically close to the user satisfies this requirement.
- (AC7) It is satisfied by DM1 due to the scalable infrastructure provided by AWS IoT whereas, in DM4, it is subject to the management of resources at the edge-level.

RQ2. To answer this question, we analyzed the impact of the use of Lambda functions for access control on performance, compared to the deployment models relying on the native access access control mechanisms provided by AWS.

AWS IoT allow the system administrator to customize and extended the native access control mechanism, hence supporting expressiveness (AC1). This can be achieved by CSP's specific functionalities, e.g., a Custom Authorizer to execute lambda functions at the cloud level (DM2-3) and the edge level (DM5-7). The increased empowerment corresponds to greater responsibilities for administrator (AC2). Functionalities like AWS Custom Authorizer, which currently lack clear documentation, need to be fully understood to avoid design flaws that could lead to the unauthorized disclosure of sensitive resources. The use of lambda function for access control enables the portability (AC3) of access control policies at both the cloud and edge level. Our experiments shows that the performances of lambda extended cloud and edge deployments are found suboptimal w.r.t. the deployment model based on native access control mechanism. We experienced high processing times and low throughput rates as compared to their respective native cloud and edge counterparts. Thus we can argue the failure of lambda extended cloud and edge deployments to satisfy the latency (AC5), reliability (AC6) and scalability (AC7) requirements.

Lambda functions are utilized at the edge level to enable the best of pure cloud and edge-based solutions i.e., support expressibility (AC1) as in DM1 and latency (AC5) as in DM4. In our work, we extended the native mechanism using Lambda function to enable a more fine-grained access control mechanism. However, the management and resource allocation concerns of Lambda functions prevented us to exploit the intended potentials.

RQ3. To answer this question, we analyzed the impact of attribute storage and retrieval i.e., the effect of policy size on the performance of the access control mechanism. The idea is to identify an optimal configuration in which the attributes are provided to the policy decision point (PDP). The possible configuration are, access request (e.g., DM2, DM5), internal storage service (e.g., DM3, DM6 and DM7) or external storage service. One remark is in place, the experiments in the cloud based deployments are only performed with native cloud storage service (i.e., AWS RDS), however, there is a possibility to fetch the attributes from an external storage service. We leave that as part of our future work.

We enhanced the expressibility of access control mechanism by enabling ABAC using Lambda functions. In cloud-based deployments, we found no impact on the performance of the access control mechanism, whereas, a slight improvement in performance is seen for deployment models in which attributes are provided from a local internal storage (local mySQL database) as compared to the an remote internal storage (e.g. AWS RDS) to a policy decision point (PDP).

Chapter 9

Related Work

In the area of access control mechanism for IoT many works have already been carried out for ensuring the security and privacy of IoT applications (see, for instance, [OMEO17b] for a compendium). However, many of these efforts have been devoted to the proposal of access control models for IoT. Nonetheless, a recent research line targets the access control enforcement for cloud-enabled IoT [CF19].

In the remainder of this section, we provide a comprehensive overview of the most relevant works related to the current stature of MQTT based IoT deployments, existing access control solutions for IoT, the analysis of the requirements of access control for IoT and metrics adapted for the performance evaluation of access control system for IoT.

9.1 Securing MQTT based IoT Deployments

Although the interest in MQTT has increased only recently, there have been some efforts to investigate its vulnerabilities and attacks; see, e.g., [ARH17, Lun17]. For the sake of brevity, we focus on the works that develop frameworks or tools to automatically detect security issues in MQTT deployments. The work in [HRVL18] describes a framework to perform template-based fuzzing of the MQTT protocol. A commercial fuzzing tool is proposed by F-Secure [FS15], in which payloads are randomly generated. While fully automated, such approaches usually require a substantial amount of time to trigger broker exceptions. The work in [AALM18] proposes IoTVerif, a tool for automatically verifying the certificates used by specific Android MQTT client applications in case MQTT brokers use TLS protocol to secure the confidentiality and integrity of exchanged messages. Our tool, MQTTSA, shares the goal of automatically identifying security problems with all these works but differ in two significant ways.

1. It focuses on different, more general, security misconfigurations in MQTT brokers that are

far from being adequately mitigated in current deployments; it is thus possible to combine the approaches in [HRVL18, FS15, AALM18] to further enrich our tool.

2. MQTTSA provides hints (at various level of details) on how to mitigate the detected security issues; this feature is not present in any other tool and we consider it as fundamental for its successful use by developers who have little or no awareness of MQTT-related security issues.

An approach closely related to ours is proposed with MQTT-PWN [AZ18]. This tool allows for automated penetration testing of MQTT brokers, including credential brute-forcing (to bypass authentication), enumeration of topics (for information gathering), the identification and extraction of sensitive information (such as passwords and GPS data). The main difference with MQTTSA are the following: MQTT-PWN does not work from the perspective of an insider attacker (i.e., when authenticating with username and password or certificates), or when the broker configured TLS; does not perform data tampering or denial of service attacks, nor returns a report containing the list of vulnerabilities and possible mitigations.

9.2 Access control solutions for smart home applications

The increasing popularity of IoT applications and, in particular, smart home applications has attracted considerable attention from the research community and industry. A large body of research has investigated the security offered by existing IoT solutions. For instance, Ur et al. [UJS13] conducted three case studies (related to lighting system, bathroom scale and door lock) to evaluate the access control systems supported by commercial smart devices. Fernandes et al. [FJP16, FRJP17] present an empirical security analysis of a leading smart home programming platform (i.e., Samsung SmartThings) that supports a broad range of devices including motion sensors, fire alarms, and door locks. Ho et al. [HLM+16] examine the security of several commercial smart lock solutions (i.e., Kevo, August, Dana, Okidokeys, Lockitron).

These studies reveal flaws in the architectural design and interaction models of commercially available smart locks and other devices, which can be exploited by an adversary to learn private information about the user and gain unauthorized access [FJP16, FRJP17, HMP+19, HLM+16], leaving users at risk for remote attacks that can cause physical, financial, and psychological harm. For instance, Ho et al. [HLM+16] show that smart locks often lack direct connectivity to the Internet due to their constrained nature. To make access control decision, smart locks typically rely on user's smart phone to interact with the centralized access control mechanism. This, however, makes smart locks vulnerable to state consistency attacks that allow an attacker to evade revocation and access logging [AMRZ18a, HLM+16]. Moreover, existing access control solutions adopted in IoT devices often fail to capture the user's understanding of access control [UJS13] and to provide usable access control specification [HGP+18]. These issues are partially

Table 9.1: Requirements for access control systems tailored to IoT environments

		Ahmad et al. [AMRZ18a]	**Ravidas et al. [RLPZ19]**	**Ouaddah et al. [OMEO17b]**	**Alonso et al. [AFMS17]**	**Tian et al. [TZL$^+$17]**
Policy Specification	Expressiveness	✓	✓	✓	✓	✗
	Extensibility	✓	✗	✗	✗	✗
	Dynamicity	✗	✓	✗	✗	✗
	Heterogeneity	✗	✗	✓	✗	✗
Policy Management	Single admin point	✓	✓	✗	✗	✗
	Flexibility	✓	✗	✓	✓	✗
	Usability	✗	✓	✓	✓	✓
	User-centric	✗	✓	✓	✗	✗
	Revocation	✗	✗	✓	✗	✗
	Delegation	✗	✗	✓	✓	✗
Policy Evaluation & Enforcement	Latency	✓	✓	✓	✗	✗
	Reliability	✓	✓	✓	✗	✗
	Scalability	✓	✓	✓	✗	✗
	Automation	✗	✓	✗	✓	✓
	Interoperability	✗	✓	✓	✗	✗
	Lightweight	✗	✗	✓	✗	✓
	Compatibility	✗	✓	✗	✗	✓

due to the limited technical understanding of smart homes and mismatch between the concerns and power of the smart home administrator (owner) and other people in the home [ZMR17].

To address these issues, several access control mechanisms for IoT and smart home applications have been proposed in the last years. These mechanisms can be broadly categorized in two groups depending on where the access control logic is deployed, namely cloud-based and edge-based. Among cloud-based solutions, Alsehri and colleagues propose in [AS16, AS17] an Access Control Oriented (ACO) architecture that extends the traditional IoT architecture (comprising object layer, middle layer, application layer) with a virtual object and a cloud service layer. Virtual objects can uniformly communicate with each other regardless of heterogeneity and locality of physical objects and the access control mechanism is used to regulate this communication as well as the communication between virtual objects and physical objects. Bhatt et al. [BPS17] propose a formal access control model for cloud-enabled IoT, called AWS-IoTAC. The feasibility of the model is shown by mapping it to the ACO architecture. Neisse et al [NSB14] propose a model-based security toolkit, called *SecKit*, for the enforcement of fine-grained security policies in MQTT brokers.

Other works propose to deploy the access control logic on the edge. Kim et al. [KBY$^+$12] propose an access control solution for seamless integration of heterogeneous devices and access control in smart homes by considering an extensible home gateway architecture. Specifically, the policy enforcement point is deployed along with a policy decision point in the home gateway. Heconsrnández-Ramos et al. [HRJMS13] propose a capability-based access control solution for IoT applications that demand real-time decision. The use of access control based on capabilities provides the security offered by cloud-based solutions in terms of validation of the issuer, subject authentication and authorization validation process but at edge level.

9.2.1 Access Control as a Service (ACaaS) for IoT

Access control mechanisms adopted by public Cloud Service Providers (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. [FMPX15] 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. [KEdHZ15] 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. [AFMS17] 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. [FAKS14] 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.

9.2.2 Requirements of access control mechanisms for IoT

The definition of access control policies in IoT is far from being a trivial process. The difficulty lies in the interpretation of complex IoT use-case specific security requirements and their translation in unambiguous and well-defined enforceable security policies. Given the complexity of IoT systems, there is a need for an access control system to accommodate all the necessary security requirements (i.e. policy specification), while maintaining a balance in terms of usage (i.e. policy management) and implementation (i.e. policy evaluation and enforcement).

A number of studies have investigated the requirements that access control solutions for IoT should meet and used these requirements as a baseline for the analysis of existing access control mechanisms for IoT. These requirements aim to identify the main concepts and design principles that have to be considered in the design and development of access control systems tailored to IoT applications [RLPZ19]. Table 9.1 presents a summary of these requirements. An access control system should be expressive enough to allow the specification of policies that can capture the security requirements of dynamically changing contextual conditions of the IoT use case. The management of policies i.e. the easiness of use and therefore applicability can be ensured by providing a single point of administration. The correct enforcement of an access control decision is always a critical issue and requires a simple and reliable access control decision in every system state.

9.3 Performance Metrics

An access control mechanism usually comes with a variety of features and administrative capabilities and, thus, their operational impact on the IoT system can be significant. NIST [HS12] provides detailed guidelines for the evaluation of access control systems. In this work, we study two of the performance properties proposed by NIST, namely *response time*, which measures the ability of an access control system to process subject requests for access within a time that is consistent with the operational needs of the use case scenario, and *policy repository and retrieval*, which aims to find a balance between hardware and software costs required for the storage of policies and attributes with efficiency based on the use case requirements.

To measure these properties, we have employed performance evaluation metrics typically used for the evaluation of cloud-based and edge-based systems. For example, Scoca et al. [SAB+18] compare edge computing with state of cloud computing solutions within the context of latency-sensitive and data-intensive applications. In their study, the quality of service (QoS) metrics experienced by the end users are evaluated in terms of network delay, processing time and service time. Bauer et al. [BA13] assess service quality of cloud-based applications from the end-user perspective by considering standards and recommendations from NIST, ISO and several other governing bodies. In particular, service quality is measured in terms of availability, latency, reliability, accessibility, retainability, throughput and timestamp accuracy. In our study, we have measured the responsiveness of access control mechanisms for IoT in various deployment models with respect latency, reliability and throughput.

Chapter 10

Conclusion

Security services for IoT ecosystem represent a key feature instrumental to foster the trust of the users and ensures the security and privacy of IoT applications. This work has focused on one of the key security service, that is, access control, by discussing the requirements that an access control solution for IoT should address, also with reference to possible deployment scenarios (i.e., Cloud and Edge based deployment models). Moreover, the requirements are validated on several use-cases of IoT and the proposed approaches (MQTTSA and ACaaS) are quantitatively verified.

We analyzed a realistic smart lock solution and identified the main requirements that access control systems for IoT should satisfy (Chap. 3). We have validated the requirements on access control (c.f. Table 3.1) for IoT solutions that we have elicited in [AMRZ18b] from the analysis of a realistic smart-lock use case scenario. We have done this by considering the variety of use case scenarios (such as container monitoring and smart metering) presented in [SGS$^+$16], a document whose main goal is to identify authorization problems. We have successfully shown that each authorization problem is covered by one (or more) of the previously identified requirements. This entitles us to conclude that the implementation of the lazy approach to ACaaS for cloud-edge IoT solutions of [AMRZ18b] can be effectively re-used in several other IoT uses cases. Indeed, qualitative and quantitative evidence that such an implementation verifies the requirements have been already provided in [AMRZ18b].

We have discussed how the choice of an architecture in the cloud-edge continuum can support the efficient and secure deployment of access control enforcement for distributed IoT systems. Our approach (Chap. 4) has been to use a combination of security analysis and the CAP theorem to understand the trade-offs underlying the choice. Crucial to make the investigation more systematic and comprehensive has been the adoption of the set of requirements identified in previous work [AR18]. For concreteness, we have applied our approach to a smart lock use case scenario which is representative of a wide range of smart home applications.

We have introduced MQTTSA (Chap. 5), a tool capable of detecting potential vulnerabilities in

MQTT brokers by automatically instantiating a set of attack patterns to expose known vulnerabilities and then generating a report describing possible mitigations. The attack patterns are extensions to the exploits and procedures described in [Lun17, HRVL18, ARH17, FBVI17]. The report contains actionable descriptions of mitigation strategies at a different level of details, ranging from narratives in natural language to code snippets that can be cut-and-paste in actual deployments. In light of our thorough experimental evaluation (Chap. 6), we believe the current version of MQTTSA is a first significant step towards assisting IoT developers in mitigating well-known (but unfortunately widely-found) vulnerabilities that result from MQTT broker misconfigurations. Being fully automated and providing actionable information for the mitigations, MQTTSA can be easily integrated within IoT deployment processes with stringent time-to-market constraints; even by developers with limited security awareness.

Driven from this analysis and the current state-of-the-art IoT platforms, we presented an ACaaS solution (Chap. 7) 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 (Chap. 8) 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. 3.1. 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., [SDV00]) 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.

10.1 Future Works

- We plan to further investigate the CAP trade-offs in context of IoT (both theoretically and experimentally). A particularly interesting line of work would be to identify appropriate synchronization protocols available in the literature or to develop new ones for the synchronization of policies and attributes in the cloud and edge.
- We plan to conduct a user study to further investigate the effectiveness of MQTTSA tool in terms of assisting developers with little security skills in adopting robust broker configurations. We also plan to expand the capabilities of the MQTTSA tool by
    1. Incorporating new attack patterns made available as security fixes or Common Vulnerabilities and Exposures (CVE)
    2. Assessing the security implications of using pre-shared keys or web-sockets.
    3. Synthesizing code snippets for other brokers besides Mosquitto.
    4. Investigate the features introduced by the latest version (5) of the MQTT standard.[1]
- We plan to extend the ACaaS tool to support more IoT platforms such as Microsoft Azure and Google Cloud Platform.

[1]http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

References

[AALM18] Khalid Alghamdi, Ali Alqazzaz, Anyi Liu, and Hua Ming. Iotverif: An automated tool to verify ssl/tls certificate validation in android mqtt client applications. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 95–102. ACM, 2018.

[Aba12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[AFMS17] Álvaro Alonso, Federico Fernández, Lourdes Marco, and Joaquín Salvachúa. Iaacaas: Iot application-scoped access control as a service. *Future Internet*, 9(4):64, 2017.

[AMRZ18a] Tahir Ahmad, Umberto Morelli, Silvio Ranise, and Nicola Zannone. A Lazy Approach to Access Control as a Service (ACaaS) for IoT: An AWS Case Study. In *Proceedings of Symposium on Access Control Models and Technologies*, pages 235–246. ACM, 2018.

[AMRZ18b] Tahir Ahmad, Umberto Morelli, Silvio Ranise, and Nicola Zannone. A lazy approach to access control as a service (acaas) for iot: An aws case study. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 235–246. ACM, 2018.

[AR18] Tahir Ahmad and Silvio Ranise. Validating Requirements of Access Control for Cloud-Edge IoT Solutions (Short Paper). In *Foundations and Practice of Security*, pages 131–139. Springer, 2018.

[ARH17] S. Andy, B. Rahardjo, and B. Hanindhito. Attack scenarios and security analysis of mqtt communication protocol in iot system. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 1–6, Sep. 2017.

[ARTW16a] A. Armando, S. Ranise, R. Traverso, and K. S. Wrona. SMT-based Enforcement and Analysis of NATO Content-based Protection and Release Policies. In *Proc. of the ABAC@CODASPY 2016*, pages 35–46, 2016.

[ARTW16b] Alessandro Armando, Silvio Ranise, Riccardo Traverso, and Konrad Wrona. SMT-based enforcement and analysis of NATO content-based protection and release policies. In *Proceedings of International Workshop on Attribute Based Access Control*, pages 35–46. ACM, 2016.

[AS16] Asma Alshehri and Ravi Sandhu. Access control models for cloud-enabled internet of things: A proposed architecture and research agenda. In *Proceedings of International Conference on Collaboration and Internet Computing*, pages 530–538. IEEE, 2016.

[AS17] Asma Alshehri and Ravi Sandhu. Access control models for virtual object communication in cloud-enabled iot. In *Proceedings of International Conference on Information Reuse and Integration*, pages 16–25. IEEE, 2017.

[AWS19a] AWS. AWS IoT. `https://aws.amazon.com/iot-core/`, 2019. Accessed: 21-Nov-2019.

[AWS19b] AWS. AWS IoT Greengrass. `https://aws.amazon.com/greengrass/`, 2019. Accessed: 21-Nov-2019.

[AWS19c] AWS. AWS Lambda. `https://aws.amazon.com/lambda/`, 2019. Accessed: 21-Nov-2019.

[AZ18] Daniel Abeles and Moshe Zioni. Mqtt-pwn documentation, 2018.

[BA13] Eric Bauer and Randee Adams. *Service quality of cloud-based applications*. John Wiley & Sons, 2013.

[BG14] Andrew Banks and Rahul Gupta. Mqtt version 3.1.1, 2014.

[BPS17] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. Access control model for aws internet of things. In *International Conference on Network and System Security*, pages 721–736. Springer, 2017.

[Bre12] Eric Brewer. Cap twelve years later: how the. *Computer*, 00(2):23–29, 2012.

[Bye17] Charles C Byers. Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled iot networks. *IEEE Communications Magazine*, 55(8):14–20, 2017.

[CF18] Pietro Colombo and Elena Ferrari. Access control enforcement within mqtt-based internet of things ecosystems. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 223–234. ACM, 2018.

[CF19] Pietro Colombo and Elena Ferrari. Access control technologies for big data management systems: literature review and future trends. *Cybersecurity*, 2(1):3, 2019.

[EMQ17] EMQ X Platform. MQTT Plugin. `https://github.com/emqtt/mqtt-jmeter`, 2017. Accessed: June 21, 2019.

[FAKS14] Paul Fremantle, Benjamin Aziz, Jacek Kopeckỳ, and Philip Scott. Federated identity and access management for the internet of things. In *Secure Internet of Things (SIoT), 2014 International Workshop on*, pages 10–17. IEEE, 2014.

[FBVI17] Syed Naeem Firdous, Zubair A. Baig, Craig Valli, and Ahmed Ibrahim. Modelling and evaluation of malicious attacks against the iot mqtt protocol. *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 748–755, June 2017.

[FJP16] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *Proceedings of Symposium on Security and Privacy*, pages 636–654. IEEE, 2016.

[FMPX15] Nikos Fotiou, Apostolis Machas, George C Polyzos, and George Xylomenos. Access control as a service for the cloud. *Journal of Internet Services and Applications*, 6(1):11, 2015.

[FRJP17] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Security implications of permission models in smart-home application frameworks. *IEEE Security & Privacy*, 15(2):24–30, 2017.

[FS15] F-Secure. A simple fuzzer for the mqtt protocol, 2015.

[FSG$^+$01] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[GL12] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[HBK18] M. S. Harsha, B. M. Bhavani, and K. R. Kundhavai. Analysis of vulnerabilities in mqtt security using shodan api and implementation of its countermeasures via authentication and acls. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2244–2250, Sep. 2018.

[HFK$^+$13] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[HGP$^+$18] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home Internet of Things (IoT). In *Proceedings of USENIX Security Symposium*, pages 255–272. USENIX Association, 2018.

[HLM$^+$16] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 461–472. ACM, 2016.

[HMP$^+$19] Weijia He, Jesse Martinez, Roshni Padhi, Lefan Zhang, and Blase Ur. When smart devices are stupid: Negative experiences using home smart devices. In *Proceedings of the SafeThings Workshop*, 2019.

[HRJMS13] José L Hernández-Ramos, Antonio J Jara, Leandro Marin, and Antonio F Skarmeta. Distributed capability-based access control for the internet of things. *Journal of Internet Services and Information Security*, 3(3/4):1–16, 2013.

[HRVL18] Santiago Hernández Ramos, M Teresa Villalba, and Raquel Lacuesta. Mqtt security: A novel fuzzing approach. *Wireless Communications and Mobile Computing*, 2018, 2018.

[HS12] Vincent C. Hu and Karen Scarfone. Guidelines for access control system evaluation metrics. NISTIR 7874, NIST, 2012.

[IEC16a] IEC. IEC Role in the Internet of Things, 2016.

[IEC16b] IEC. IoT 2020: Smart and Secure IoT Platform, 2016.

[KBY$^+$12] Ji Eun Kim, George Boulos, John Yackovich, Tassilo Barth, Christian Beckel, and Daniel Mosse. Seamless integration of heterogeneous devices and access control in smart homes. In *Proceedings of International Conference on Intelligent Environments*, pages 206–213. IEEE, 2012.

[KEdHZ15] Samuel Paul Kaluvuri, Alexandru Ionut Egner, Jerry den Hartog, and Nicola Zannone. Safax–an extensible authorization service for cloud environments. *Frontiers in ICT*, 2:9, 2015.

[Kle15] Martin Kleppmann. A critique of the cap theorem. *arXiv preprint arXiv:1509.05393*, 2015.

[Loc10] Dave Locke. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks Technical Library*, page 15, 2010.

[Lun17] Lucas Lundgren. Taking over the world through mqtt—aftermath, 2017.

[Met07] LLC Metasploit. The metasploit framework, 2007.

[Mon17] Nolan Mondrow. Lockstate 6i/6000i update, August 2017.

[MR17] Umberto Morelli and Silvio Ranise. Assisted authoring, analysis and enforcement of access control policies in the cloud. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 296–309. Springer, 2017.

[MVQ18] Federico Maggi, Rainer Vosseler, and Davide Quarta. The Fragility of Industrial IoT's Data Backbone—Security and Privacy Issues in MQTT and CoAP Protocols, 2018.

[NSB14] Ricardo Neisse, Gary Steri, and Gianmarco Baldini. Enforcement of security policy rules for the internet of things. In *Proceedings of International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 165–172. IEEE, 2014.

[OMEO17a] Aafaf Ouaddah, Hajar Mousannif, Anas Abou Elkalam, and Abdellah Ait Ouahman. Access control in the internet of things: Big challenges and new opportunities. *Computer Networks*, 112:237–262, 2017.

[OMEO17b] Aafaf Ouaddah, Hajar Mousannif, Anas Abou Elkalam, and Abdellah Ait Ouahman. Access control in the internet of things: Big challenges and new opportunities. *Computer Networks*, 112:237–262, 2017.

[PAG$^+$18] Pasquale Pace, Gianluca Aloi, Raffaele Gravina, Giuseppe Caliciuri, Giancarlo Fortino, and Antonio Liotta. An edge-based architecture to support efficient applications for healthcare industry 4.0. *IEEE Transactions on Industrial Informatics*, 15(1):481–489, 2018.

[PPR$^+$19] Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, and Tahir Ahmad. Mqttsa: A tool for automatically assisting the secure deployments of mqtt brokers. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642, pages 47–53. IEEE, 2019.

[PVPG17] Giovanni Perrone, Massimo Vecchio, Riccardo Pecori, and Raffaele Giaffreda. The day after mirai: A survey on mqtt security solutions after the largest cyber-attack carried out through an army of iot devices. In *IoTBDS*, pages 246–253, 2017.

[RCK$^+$19] Gowri Sankar Ramachandran, Sharon LG Contreras, Bhaskar Krishnamachari, Ulas C Kozat, and Yinghua Ye. Publish-pay-subscribe protocol for payment-driven edge computing. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[RLPZ19] Sowmya Ravidas, Alexios Lekidis, Federica Paci, and Nicola Zannone. Access control in Internet-of-Things: A survey. *Journal of Network and Computer Applications*, 2019.

[SAB$^+$18] Vincenzo Scoca, Atakan Aral, Ivona Brandic, Rocco De Nicola, and Rafael Brundo Uriarte. Scheduling latency-sensitive applications in edge computing. In *Proceedings of International Conference on Cloud Computing and Services Science*, pages 158–168. SciTePress, 2018.

[SCW10] Andy J Stanford-Clark and Glenn R Wightwick. The application of publish/subscribe messaging to environmental, monitoring, and control systems. *IBM Journal of Research and Development*, 54(4):1–7, 2010.

[SDV00] Pierangela Samarati and Sabrina Capitani De Vimercati. Access control: Policies, models, and mechanisms. In *International School on Foundations of Security Analysis and Design*, pages 137–196. Springer, 2000.

[SGS$^+$16] L Seitz, S Gerdes, G Selander, M Mani, and S Kumar. Use cases for authentication and authorization in constrained environments. *RFC 7744*, 2016.

[SMG15] Stavros Salonikias, Ioannis Mavridis, and Dimitris Gritzalis. Access control issues in utilizing fog computing for transport infrastructure. In *International Conference on Critical Information Infrastructures Security*, pages 15–26. Springer, 2015.

[SRSB15] Meena Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar. Secure mqtt for internet of things (iot). In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 746–751, April 2015.

[Sta13] OASIS Standard. extensible access control markup language (xacml) version 3.0. 22 january 2013, 2013.

[TCH16] William Tärneberg, Vishal Chandrasekaran, and Marty Humphrey. Experiences creating a framework for smart traffic control using aws iot. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 63–69. ACM, 2016.

[TdRZ17] Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. Formal analysis of XACML policies using SMT. *Computers & Security*, 66:185–203, 2017.

[TZL$^+$17] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *Proceedings of USENIX Security Symposium*, pages 361–378. USENIX Association, 2017.

[UJS13] Blase Ur, Jaeyeon Jung, and Stuart Schechter. The current state of access control for smart devices in homes. In *Proceedings of Workshop on Home Usable Privacy and Security*, 2013.

[XHF$^+$17] Xiaomin Xu, Sheng Huang, Lance Feagan, Yaoliang Chen, Yunjie Qiu, and Yu Wang. Eaaas: Edge analytics as a service. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 349–356. IEEE, 2017.

[YV00] Haifeng Yu and Amin Vahdat. Building replicated internet services using tact: A toolkit for tunable availability and consistency tradeoffs. In *Proceedings Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems. WECWIS 2000*, pages 75–84. IEEE, 2000.

[ZMR17] Eric Zeng, Shrirang Mare, and Franziska Roesner. End user security and privacy concerns with smart homes. In *Proceedings of Symposium on Usable Privacy and Security*, pages 65–80. USENIX Association, 2017.