Internet of Things: A survey on the Security of IoT Frameworks

Mahmoud Ammar[1], Giovanni Russello[2], and Bruno Crispo[1]

[1]Department of Computer Science, KU Leuven University, Heverlee, 3001, Belgium
E-mail: {firstname.lastname}@cs.kuleuven.be
[2]Department of Computer Science, University of Auckland, Private Bag 92019, Auckland 1142, New Zealand
E-mail: g.russello@auckland.ac.nz

***Abstract*—The Internet of Things (IoT) is heavily affecting our daily lives in many domains, ranging from tiny wearable devices to large industrial systems. Consequently, a wide variety of IoT applications have been developed and deployed using different IoT frameworks. An IoT framework is a set of guiding rules, protocols, and standards which simplify the implementation of IoT applications. The success of these applications mainly depends on the ecosystem characteristics of the IoT framework, with the emphasis on the security mechanisms employed in it, where issues related to security and privacy are pivotal. In this paper, we survey the security of the main IoT frameworks, a total of 8 frameworks are considered. For each framework, we clarify the proposed architecture, the essentials of developing third-party smart apps, the compatible hardware, and the security features. Comparing security architectures shows that the same standards used for securing communications, whereas different methodologies followed for providing other security properties.**

***Index Terms*—Internet of Things, IoT, framework, platform, security.**

I. Introduction

The Internet of Things (IoT) plays a remarkable role in all aspects of our daily lives. It covers many fields including healthcare, automobiles, entertainments, industrial appliances, sports, homes, etc. The pervasiveness of IoT eases some everyday activities, enriches the way people interact with the environment and surroundings, and augments our social interactions with other people and objects. This holistic vision, however, raises also some concerns, like which level of security the IoT could provide? and how it offers and protects the privacy of its users?

Developing applications for the IoT could be a challenging task due to several reasons; (i) the high complexity of distributed computing, (ii) the lack of general guidelines or frameworks that handle low level communication and simplify high level implementation, (iii) multiple programming languages, and (iv) various communication protocols. It involves developers to manage the infrastructure and handle both software and hardware layers along with preserving all functional and non-functional software requirements. This complexity has led to a quick evolution in terms of introducing IoT programming frameworks that handle the aforementioned challenges.

Very recently, several IoT frameworks have been launched by the major shareholders in the IoT domain and by the research community in order to support and make it easy to develop, deploy and maintain IoT applications. Each player built his approach depending on his vision towards the IoT world [1]. In this survey, we compare the properties of a subset of IoT frameworks, targeting in particular their security features. The selected set of IoT platforms[1] includes: *AWS IoT* from *Amazon*, *ARM Bed* from *ARM* and other partners, *Azure IoT Suite* from *Microsoft*, *Brillo/Weave* from *Google*, *Calvin* from *Ericsson*, *HomeKit* from *Apple*, *Kura* from *Eclipse*, and *SmartThings* from *Samsung*.

We selected the above frameworks based on the following criteria: (i) the reputation of the vendors in the software and electronics industries, (ii) the support of rapid application development and the number of applications on the store, (iii) the coverage and usage of the framework, and it's popularity in the IoT market.

The objectives of this survey are manifold:

- Giving a picture of the current state of the art IoT platforms and identifying the trends of current designs of such platforms.
- Providing a high level comparison between the different architectures of the various frameworks.
- Focusing on the models designed and approaches developed for ensuring security and privacy in these frameworks.
- Illustrating the pros and cons of each framework in terms of fulfilling the security requirements and meeting the standard guidelines.
- Exploring the design flaws and opening the door for more in depth security analysis against potential threats.

The remainder of this paper is structured as follows: Section II describes the general concept of the IoT framework. Related works are presented in Section III. Section IV is the backbone of this paper which provides a horizontal overview of the various IoT frameworks and focuses on the related security features. A discussion is provided in Section V. Finally, Section VI concludes this study.

[1]In this paper, the terms *Framework* and *Platform* are used interchangeably.

II. Background

The very rapid growth of Internet-connected devices, ranging from very simple sensors to highly complex cloud servers, shapes the ***Internet of Things***, where *Things*, in this context, refers to a wide variety of objects (e.g. smart bulbs, smart locks, IP cameras, thermostats, electronic appliances, alarm clocks, vending machines, and more). The resemblance between all IoT objects is the ability to connect to the Internet and exchange data. The network connectivity feature allows controlling objects remotely across the existing network infrastructure, resulting in more integration with the real world and less human intervention. The IoT transforms these objects from being classical to smart by exploiting its underlying technologies such as pervasive computing, communication capabilities, Internet protocols, and applications. Protocols are required in order to identify the spoken language of the IoT devices in terms of the format of exchanged messages, and select the correct boundaries that comply with the various functionality of each device. Applications determine levels of granularity and specialty of the IoT device and how big are the data generated for analytics purposes. They also indicate the general scope of the IoT framework covering the context of the applied domain.

The concept of IoT framework entails identifying a structure which coordinates and controls processes being conducted by the various IoT elements. This structure is a set of rules, protocols and regulations that organize the way of processing data and exchange messages between all involved parties (e.g. embedded devices, cloud, end-users). Also, it should support the high level implementation of IoT applications and hide the complexity of infrastructure protocols. There are several approaches that can be followed to build an IoT framework depending on the requirements of the target business [2].

In this survey, we are targeting IoT frameworks based on the public cloud approach, as they are the most commonly used and widely available in the IoT market. The main building blocks of any cloud-based IoT framework are the physical objects and the protocols. Physical objects include: (i) smart devices such as sensors, actuators, etc., (ii) servers act as a cloud-backend or hubs/gateways for routing, storing, and accessing various pieces of data, and (iii) end-users represented by the applications they use to access data and interact with IoT devices. Protocols run on different layers and provide end-to-end communication. To the best of our knowledge, there is no a standard IoT architecture yet. For simplicity, we are considering the basic one which is a 3-layer architecture [3] composed of *Application*, *Network*, and *Perception* layers. The *Perception* layer belongs to the physical devices that identify and sense analog data and then digitize it for transportation purposes. Infrastructure protocols such as ZigBee [4], Z-Wave [5], Bluetooth Low Energy (BLE) [6], WiFi, and LTE-A [7] run in the *Network* layer. The *Application* layer is the interface for end-users to access data and talk to their IoT devices. It supports standard protocols such as *Hyper Text Transfer Protocol* (HTTP) [8], *Constrained Application Protocol* (CoAP) [9], *Message Queue Telemetry Transport* (MQTT) [10], *Extensible Messaging and Presence Protocol* (XMPP) [11], *Advanced Messaging Queuing Protocol* (AMQP) [12], and *Data Distribution Service* (DDS) [13].

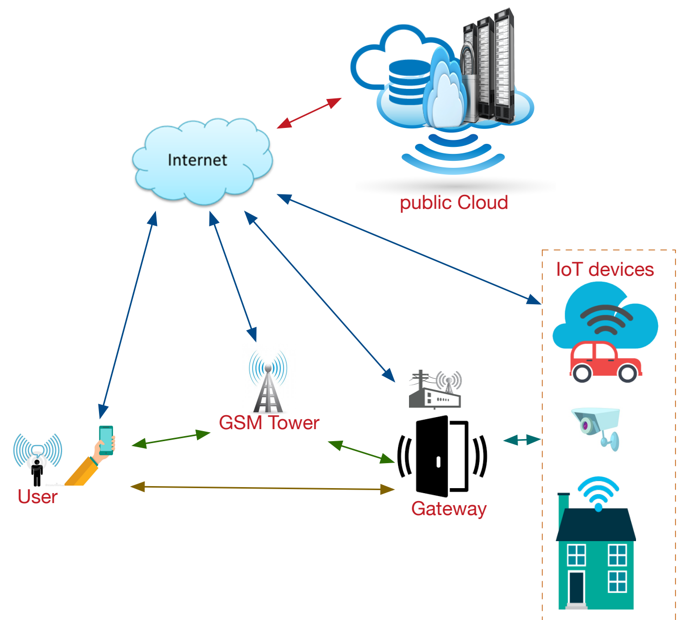

Fig. 1: A high level system model of IoT

The system model, presented in Figure 1, helps to gain a better insight into the real meaning of IoT, and understand the importance of having a framework, in which, hiding the complexity and bringing simplicity to application development are axial. The IoT framework should handle the life cycle of sensing, computing, delivering, and presenting data. Depending on their capabilities, some IoT devices can reach the outside world (e.g. the cloud) directly and some others must connect to a hub or a gateway in order to connect to the external world. For the IoT frameworks considered in this survey, the cloud is the backbone, which offers databases for storing data, services for data analytics, security modules for preserving confidentiality and supports privacy, and other services. Customers use their smart phones, tablets, or laptops to interact with other IoT devices indirectly through either a cloud backend or a gateway.

In spite of targeting the same objective, different approaches have been designed and followed by vendors in order to build their IoT frameworks. In particular, the following questions arise regarding the design details of such frameworks:

- How each IoT framework handles the communication processes between IoT devices and cloud? Between cloud and end-users? What are the protocols and techniques used?
- What are the hardware and software dependencies in each framework?
- To which extents these frameworks use the common security standards?
- What are the security-related functionality offered by each element/layer in each IoT framework?
- How each framework solves the challenge of preserving

security and privacy among all involved parties? what are the techniques used for providing authentication, authorization, access control, cryptography, and other security features?

Section IV answers the above questions for each framework considered in this study.

III. Related Work

Several survey papers have been published covering various topics of the IoT domain. *Al-Fuqaha et al.* [14] surveyed the IoT in general, mentioning various IoT architectures, market opportunities, IoT elements, communication technologies, standard application protocols, main challenges and open research problems in the IoT area. *Derhamy et al.* [2] presented a number of commercial IoT frameworks and provided a comparative analysis based on utilized approaches, supported protocols, usage in industry, hardware requirements, and applications development. A brief overview of the current IETF standards for the Internet of things is provided in [15].

Security and privacy issues in IoT had a lot of attention by the research community and addressed at different levels. In [16], the authors surveyed the security and privacy issues in IoT from four different perspectives. First, they highlight on the limitations of applying security in IoT devices (e.g. battery lifetime, computing power) and the proposed solutions for them (e.g. lightweight encryption scheme designed for embedded systems). Second, they summarize the classifications of IoT attacks (e.g. physical, remote, local, etc.). Third, they focus on the mechanisms and architectures designed and implemented for authentication and authorization purposes. Last, they analyse the security issues at different layers (e.g. physical, network, etc.). Authors in [17], [18] addressed the security and privacy issues in IoT at each layer identified in the 3-layer architecture [3]. [20] surveyed most of the security flaws existing in IoT, resulted from the various communication technologies used in wireless sensor networks. An authorization access model is proposed in [21] as a security framework for the IoT in order to ensure controlling access and authorizing legitimate users only. Authors in [22] reviewed the challenges and approaches proposed to overcome the security issues of the IoT middleware, where a large number of existing systems inherit security properties from the middleware frameworks. Depending on the well-known security and privacy threats, authors analyse and evaluate the available middleware approaches and show how security is handled by each approach. The work concludes with illustrating a set of requirements to have a secure IoT middleware.

All of the aforementioned surveys review the IoT security with regards to one element of the common IoT standards (e.g. network protocols or middleware employed). To the best of our knowledge, this survey is the first one of addressing the IoT security at the programming level by evaluating the security features of a subset of commercially available IoT programming frameworks.

IV. IoT Frameworks

A. AWS IoT

AWS (Amazon Web Services) IoT [23] is a cloud platform for the Internet of things released by Amazon. This framework aims to let smart devices easily connect and securely interact with the AWS cloud and other connected devices. With *AWS IoT*, it is easy to use and utilize various AWS services like Amazon DynamoDB [24], Amazon S3 [25], Amazon Machine Learning [26], and others. Furthermore, *AWS IoT* allows applications to talk with devices even when they are offline.[2]

1) Architecture: As shown in Figure 2, the *AWS IoT* architecture consists of four major components: the *Device Gateway*, the *Rules Engine*, the *Registry*, and the *Device Shadows* [27].

The *Device Gateway* acts as an intermediary between connected devices and the cloud services, which allows these devices to talk and interact over the MQTT protocol. In spite of being an old protocol, in comparison with other IoT protocols, Amazon uses MQTT [10] due to several features; (i) fault tolerance property, (ii) excellent for intermittent connectivity, (iii) small footprint in terms of the space needed in the device memory, (iv) very efficient in terms of the network bandwidth requirements, and (v) depends on the publish/subscribe programming model to allow one-to-many communication between various devices [28]. The latter feature means that sensors and other embedded devices that are moving and talking to the *Device Gateway* do not need to know who is sending data to them. They just send the data route and those who subscribe to the data will receive it. This enables a scalable environment for low-latency, low-overhead, and bi-directional communication. Under the hood, the *Device Gateway* is built in a fully managed and highly available environment controlled by the community of Amazon 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 MQTT messages are carried out over TLS (*Transport Layer Security*), the successor of SSL (*Secure Socket Layer*) [29]. Furthermore, the *Device Gateway* supports WebSockets and HTTP 1.1 protocols [30].

On the other hand, the *Device Gateway* is teamed up with another component called *Rules Engine*. The *Rules Engine* processes incoming published messages and then transforms and delivers them to other subscribed devices or AWS cloud services, as well as to non-AWS services via *AWS Lambda* [31] for further processing or analytics. This enables the possibility to build IoT applications that orchestrate, collect, process, analyze, and act on data generated and published by connected devices globally without having to pay attention to the low level network protocols or manage any infrastructure. In order to maintain usability, developers can author rules and add them to the *Rules Engine* by writing SQL-like statements or using

[2] Using *Device Shadows* as discussed later in the Architecture.

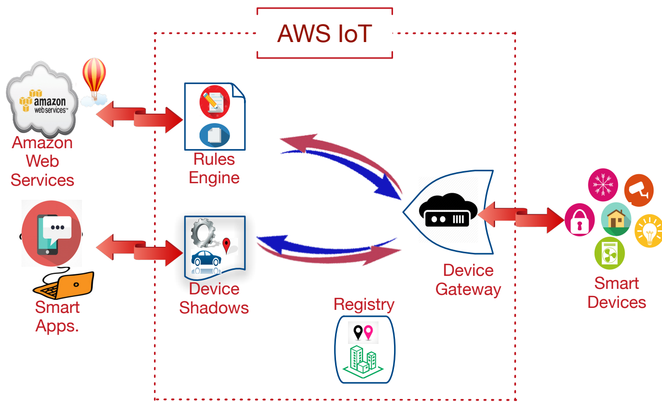

Fig. 2: AWS IoT Architecture

the *AWS Management Console* service [32]. Considering the example shown in Listing 1[3], the rule consists of two main segments: the SQL statement and the actions list. The SQL statement identifies the publish/subscribe topics to apply the rule on, and the conditions under which the rule should be executed. The actions list specifies a set of actions that should be performed when the SQL statement is executed. The rule definitions use a JSON-based schema.

```
{
  "sql": SELECT * FROM 'iot/tempSensors/#' WHERE
      temp >50,
  "actions":[
   {
    "dynamoDB":{
      "tableName":"HighTempTable",
      "roleArn":"arn:aws:iam::
          your-aws-id:role/dynamoPut",
      "hashKeyField":"key",
      ...
    }
   }
  ]
}
```

Listing 1: Example of a defined rule in the Rules Engine

Rules behave differently depending on the content of each incoming message. Apart from this, the *Rules Engine* offer dozens of built-in helping functions and calculations to aggregate, transform, concatenate, and process data and build very sophisticated rules. Developers can create their own functions and define others using *AWS Lambda*. the *Rules Engine* can receive data from multiple sources, different devices, and even from the AWS cloud. It integrates and routes this information to other IoT devices and AWS cloud services such as *Amazon Kinesis* [33], *Amazon S3*, *Amazon DynamoDB*, etc.

The *Registry* unit is responsible for assigning a unique Id to each connected device regardless the device type, vendor, or the way of connection. Also, it stores the metadata (e.g. device name, Id, attributes, etc.) of connected devices in order to have the capability of tracking them. If the device is not active anymore and did not show up in the network for a period of 7 years, the metadata will be expired and removed from the *Registry*. Either *AWS IoT Management Console* or the *AWS Command Line Interface* [34] can be used to interact with the *Registry* and configure it manually.

[3]This example has been taken from the online Amazon tutorials.

AWS IoT instantiates each connected device by creating a virtual image called *Device Shadow*. This *shadow* is persistent and stored in the cloud to be available and accessible all the time. It represents the last state of the device when it was online, and enforces the future state over the physical device once it shows up again in the network. This means that cloud services and other devices can integrate, communicate, and read the current state of a certain device through its *shadow* even if the device is offline. They can update the state of the device as well. Updates are applied once the device gets online. Reading the last reported state and setting the desired future state is done by interacting with *Device Shadows* via REST API or by using the *Rules Engine*. This functionality helps in easily controlling devices and performing actions over them without having to know about the low level of connectivity. This means that the *shadow* accelerate applications development by providing a uniform and available interface to devices, even when they use different IoT communication and security protocols, or even when they are constrained by intermittent connectivity, limited bandwidth, limited computing ability, or limited power. From a programming point of view, the *Device Shadow* is a JSON document, which used to store and retrieve the current state of a certain device.

Optionally, applications can communicate directly to the connected physical devices using only the *Device Gateway and the Rules Engine*. This means ignoring the *Registry and Device Shadow*. Nevertheless, it is not recommended since the user has to focus on maintaining the underlying communication protocols and solving synchronization issues between the connected devices and the cloud.

AWS IoT provides a *Device SDK* which makes it easy for the device to synchronize its state with its *shadow*, and accept the desired future states. In particular, The *AWS IoT Device SDK* is a set of libraries to help connecting hardware devices, authenticating with the cloud, installing mobile applications, and exchanging messages easily. It supports different programming languages including C and JavaScript.

2) Smart Applications Specifications: The *AWS IoT* has no restrictions regarding either the programming languages of developing smart applications or operating systems running them. Users can use various platforms (e.g. mobiles, laptops, etc.) to interact with their cloud-connected IoT devices via REST APIs. In general, there are two types of smart applications in *AWS IoT*;*companion* and *server* apps. The latter are designed and implemented to monitor, manage, and control a large number of connected devices at the same time. An example of a server application would be a fleet management website that plots thousands of trucks on a map in real-time. Companion apps are mobile or web-based applications that allow end-users to interact with their cloud-connected devices. As stated previously, companion and servers apps can access and communicate with *device shadows* in the cloud via uniform Restful APIs.

3) Hardware Specifications: AWS IoT provides an open-source client libraries and device SDKs that make the framework available for several embedded operating systems and microcontroller platforms. To the best of our knowledge, the device SDKs supports C, Node.js, and the Arduino platform. Any IoT device can connect to the *AWS IoT cloud* if it has the ability to be configured using one of the aforementioned programming languages. Even those devices that connect to private IP networks or communicate using non-IP protocols, e.g. ZigBee, can access the *AWS IoT cloud* as long as they are connected to a physical hub, which serves as an intermediary gateway for the external world (e.g. AWS cloud).

4) Security Features: Amazon leverages a multi-layer security architecture for the *AWS IoT*, in which, the security is applied at every level of the technology stack. The design of the security architecture is based on teaming up the *Message Broker* service with the *Security and Identity* service as shown in Figure 3 [4].

- **Authentication:** In order to connect a new IoT device to the *AWS IoT Cloud*, the device has to be authenticated. The *AWS IoT* supports mutual authentication at all points of connection, so that the source of the transmitted data is always known. In general, *AWS IoT* provides three ways of verifying identity:
  - X.509 certificates [35].
  - AWS IAM users, groups, and roles [36].
  - AWS Cognito identities [37].

  The most commonly technique used for authentication, in *AWS IoT*, is X.509 certificates [38]. They are digital certificates, depend on the public key cryptography, and should be issued by a trusted party called *a certification authority* (CA). In our case, the *security and identity* unit in the *AWS IoT cloud* acts as a CA. These certificates are SSL/TLS-based to ensure secure authentication. Utilizing the authentication mode in the SSL/TLS protocol, *AWS IoT* verifies the certificate of any object by asking the client for his ID (e.g. AWS account) along with the corresponding X.509 certificate to check validity against a registry of certificates. *AWS IoT* then challenges the client to prove the ownership of the private key that belongs to the public key provided in the certificate. Optionally, the user can use his own certificate issued by his preferred CA. However, he must register this certificate in the registry.

  HTTP and WebSockets requests sent to the *AWS IoT* are authenticated using either *AWS Identity and Access Management* (AWS IAM) [39] or *AWS Cognito* [40]. Both of which support the AWS method of authentication. It's called *AWS Signature Version 4* (SigV4) [41]. For HTTP protocol, it is optional to use one of these methods for authentication, but using MQTT requires authenticating using only X509 certificates. In contrast, connection using WebSockets is limited only to the use of SigV4 for authentication.

[4]The figure has been taken from Amazon documentation.

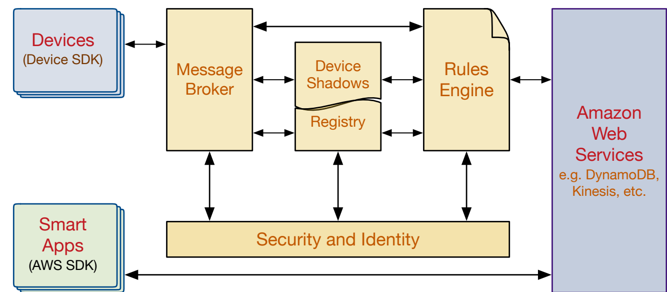

Fig. 3: AWS IoT Security Mechanism

  To sum up, each IoT device, connected to the *AWS IoT*, is authenticated using one of the methods discussed, chosen by the end-user. It is the responsibility of the message broker to authenticate and authorize all actions in the user's account. In particular, it is responsible to authenticate all attached devices, securely ingest device data, and adhere to the access permissions applied by the user on his devices using policies.

- **Authorization and Access Control:** The authorization process in *AWS IoT* is policy-based. It can be applied by either mapping authored rules and policies to each certificate or applying IAM policies. This means that only devices or applications specified in these rules can have access to the corresponding device, that this certificate belongs to. This can be ensured by the use of the *Rules Engine* since the communication through *AWS IoT* follows the principle of least privilege. The *Rules Engine* has the responsibility to leverage the AWS access management system to securely access and transfer data to its final destination according to the predefined rules/policies. So, the owner of a cloud-connected device can write some rules in the *Rules Engine* to authorize some devices or applications to access his device and prevent others. The use of AWS policies or IAM policies offers a complete control over own devices and regulates other's right to access their capabilities and perform operations over them [42].

- **Secure Communication:** All traffic to and from *AWS IoT* is encrypted over SSL/TLS protocol. TLS is used to ensure the confidentiality of the application protocols (MQTT, HTTP) supported by *AWS IoT*. For both protocols, TLS encrypts the connection between the device and the *Message Broker*. Many TLS cipher suites are supported in *AWS IoT* including: ECDHE-ECDSA-AES128-GCM-SHA256, AES128-GCM-SHA256, AES256-GCM-SHA384, etc. Furthermore, *AWS IoT* supports *Forward Secrecy*, a property of secure communication protocols, in which compromising long-term keys does not compromise temporary session keys. This means that a malicious user who learns the private key of an IoT device should not be able to decrypt any communication protected under this key unless learning the temporary key of each session.

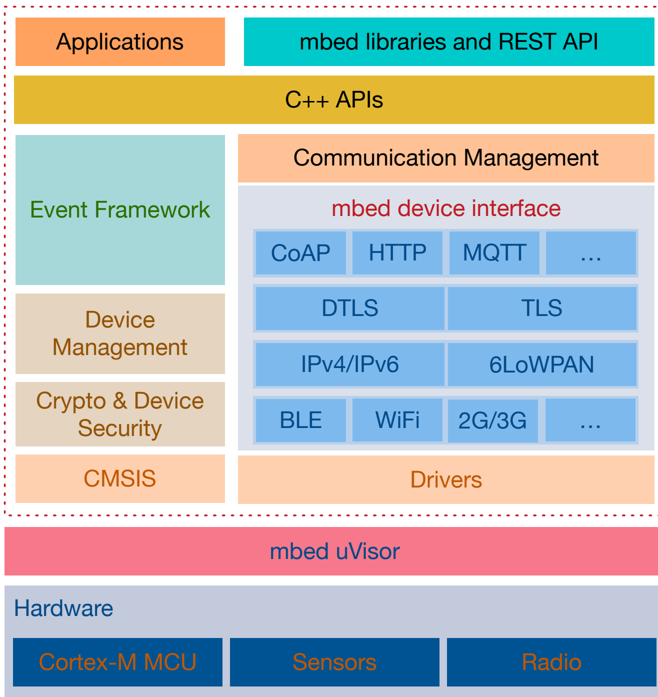

Fig. 4: mbed OS Architecture

AWS IoT cloud assigns a private home directory for each legitimate user. All private data are stored encrypted using symmetric key cryptography (e.g. AES128).

B. ARM mbed IoT

ARM mbed IoT is a platform to develop applications for the IoT based on ARM microcontrollers [43]. It provides all requirements through it's ecosystem to build either an IoT standalone applications or networked ones [44]. *ARM mbed IoT* platform aims to provide scalable, connected, and secure environment for IoT devices by integrating mbed tools and services, ARM microcontrollers, *mbed OS*, *mbed Device Connector*, and *mbed Cloud*.

ARM mbed IoT framework has the advantage over the vast majority of frameworks by providing a common OS foundation for developing IoT. It supports the most important communication protocols for connecting devices with each others and with the cloud. Furthermore, it supports automatic power management in order to solve the power consumption problem.

1) Architecture: The key building blocks of the *ARM mbed IoT* platform are *mbed OS*, *mbed client library*, *mbed cloud*, *mbed device connector*, and hardware devices based on ARM microcontrollers. The *mbed OS* represents the backbone of this platform. Therefore, discussing it's architecture helps in simplifying the architecture of the *ARM mbed IoT* platform and clarifying it.

ARM *mbed OS* [45] is an open source and full stack operating system designed for embedded devices, specifically, ARM Cortex-M microcontrollers, used to power smart homes and smart cities. It is built in a modular fashion, so that developers can use it as a complete operating system or just pick what meets their needs from its modules. The *mbed OS* represents the device-side component and stands on the top of a device security module, called *mbed uVisor*.

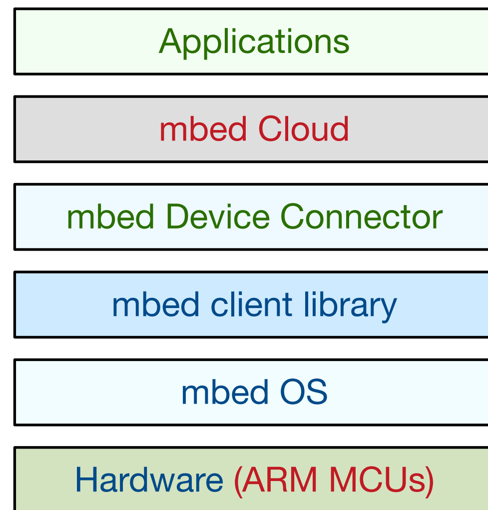

Fig. 5: ARM mbed IoT Architecture

Figure 4 presents the various modules of the architecture of the *mbed OS*. It is an event-driven architecture and does not support multi-threaded environment. *mbed OS* provides a core operating system, drivers that simplify the connectivity with the hardware layer, security and device management functionality, a suite of standard communication protocols, and multiple APIs for integration and interaction purposes.

The *mbed device interface* layer supports a wide variety of communication protocols including Bluetooth low energy (BLE), WiFi, Ethernet, ZigBee IP, 6LoWPAN, and many others. In particular, the TLS/DTLS sub-layer represents *mbed TLS* security module and ensures the end-to-end security across the communication channels. Also, multiple application protocols are supported in the architecture such as CoAP, HTTP, and MQTT.

mbed OS is designed to work in concert with *mbed Device Connector*, *mbed Device Server*, and *mbed Client*. Together, they form the platform that delivers comprehensive IoT solutions.

A high level view of the *mbed IoT* architecture is provided in Figure 5. The hardware layer, at the base, represents *mbed IoT*-enabled devices. One level up, the *mbed OS* takes a place with all it's components.

The *mbed client Library* is the key to communicate with the upper layer in the architecture. In particular, it encapsulates a subset of the *mbed OS* functionality in order to be able to connect physical devices to the *mbed Device Connector Service*. Practically, the *mbed Client Library* is a C++ API which implements a communication stack with low power consumption based on CoAP, and supports security measures (e.g. mbed TLS) that comply with constrained networks and devices. Furthermore, it is portable to various operating systems (e.g. RTOS and Linux) and supports OMA Lightweight

Machine to Machine (LWM2M) compliance [46].

The *mbed Device Connector* is a web service that helps developers to connect IoT devices to the cloud without taking care of the infrastructure [47]. It is full compatible with the *mbed OS* and can be accessed via the *mbed Client Library*. Also, it works with REST APIs, making it easy to integrate and transit to the various commercial service providers. Moreover, the *mbed Device Connector* provides end-to-end trust and security using TLS/DTLS, and supports a wide range of standard protocols including CoAP/HTTP, TLS/TCP, DTLS/UDP and OMA Lightweight M2M.

Recently, ARM community announced about *mbed Cloud* [48], and integrated it into the IoT ecosystem. It is a Software as a Service (SaaS) solution for managing IoT devices. The *mbed Cloud* allows users to securely update, provision, and connect devices. It aims to provide all security guarantees in terms of cryptography modules, trusted zones, keys management, etc. Because of being a SaaS, the *mbed Cloud* can be shipped out and configured by end users depending on their business needs. In practice, the *mbed Device Connector* is a hosted instance of the mbed Cloud services.

The top layer of the *mbed IoT* architecture is the third-party applications. Developers can implement various web and smart applications to manage cloud-connected IoT devices via REST API.

2) Smart Applications Specifications: Using the *mbed IoT* platform involves implementing embedded applications for IoT devices as well as smart apps for end-users. Developers have to use C++ programming language at the device side. At the user side, there is no prior requirements, any programming language supports REST API can be used (e.g. Java) [49].

3) Hardware Specifications: ARM mbed IoT platform is mainly dedicated to ARM Cortex-M based 32-bits[5] microcontrollers supporting advanced RISC architecture. Other microcontrollers are not supported.

4) Security Features: The security architecture of *mbed IoT* platform is applied at three different levels:

- The device itself (as a hardware & mbed OS).
- The communication channels.
- The lifecycle of developing embedded and smart applications in terms of device management, firmware updates, etc.

Figure 6 provides an overview of the security architecture [50]. The core components are:

- The *mbed uVisor* [51]: the device-side security solution, which has the ability to isolate various pieces of software from each others and from the operating system.

[5] 8-bits and 16-bits architectures can be used without selecting security modules in mbed OS.

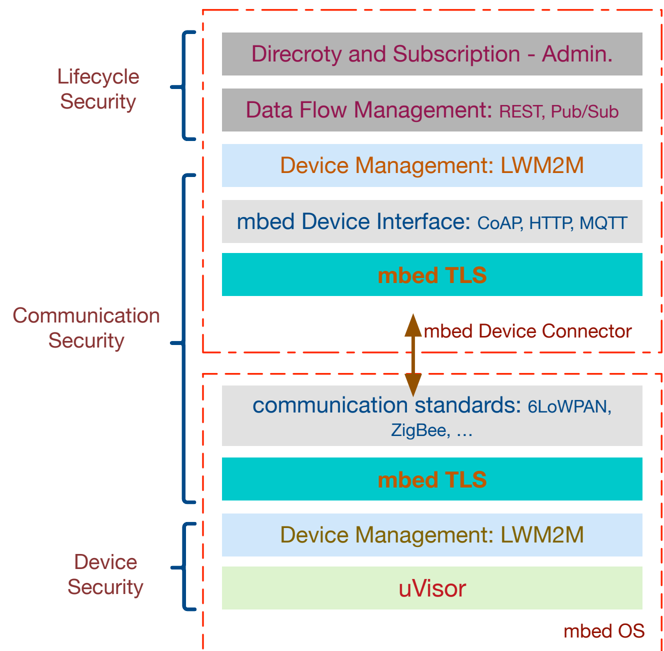

Fig. 6: ARM mbed IoT Security Architecture

- The *mbed TLS* [52]: for securing communication, confidentiality, and authentication purposes.

The following security properties are provided by the aforementioned security components.

- **Authentication:** There is no specific way of authentication. *ARM mbed IoT* provides a wide variety of cryptography standards, key exchange mechanisms, certificate-based signatures, and symmetric and public/private key encryptions through the *mbed TLS* software block [52]. Developers can pick from this basket what is suitable for them to perform the authentication process efficiently e.g. X.509 certificates.

- **Authorization and Access Control:** Arm mbed IoT devices support multiprogramming, so memory is not a single unprotected space, but it's organized into compartmentalized blocks, resulting in good security levels. Therefore, in order to control access to resources and preserve levels of authorization, the *mbed IoT* platforms depends on the ARMv7-M architecture in terms of having *MPU* and *uVisor* components.

  The *Memory Protection Unit* (MPU) is a hardware module, which enforces memory isolation. The *uVisor* is a self-contained software hypervisor, which represents the basis of the kernel of mbed OS security architecture. It acts as a sandbox and uses the MPU to enforce isolated security domains within the microcontroller itself (Cortex-M3, M4, or M7). Forming isolated domains protect sensitive parts of the system, as each part is located in a different portion of the memory. In other words, the application will be composed of some non-intersected sections. Attacking any section does not

violate others. Moreover, having any bug or security flaw in some sections of the system does not threaten others.

In summary, the *uVisor* secures software running on Cortex-M3, Cortex-M4, and Cortex-M7 processors by segmenting memory into insecure (public) and secure (private) memory spaces based on the MPU.

- **Secure Communication:** End-to-end security is ensured between all involved parties in the communication channel by implementing the TLS/DTLS protocol. It is the cornerstone of securing all communications.
  In *mbed OS*, the *mbed TLS* provides security mechanism in order to secure and protect communication, by supporting Transport Layer Security (TLS) and the related Datagram TLS (DTLS) protocol. Both protocols are the state of the art standards for securing communication over the World Wide Web. This means preventing eavesdropping, tampering and message forgery and ensuring integrity.
  The *mbed TLS* also includes reference quality software implementations of a wide range of popular cryptographic primitives, secure key management, certificate handling, and other cryptographic functionality. In addition, ARM benefits from the hardware cryptography block in some microcontrollers to encrypt sensitives parts of data.

C. Azure IoT Suite

Microsoft has released *Azure IoT Suite* [53], a platform composes of a set of services that enable end-users to interact with their IoT devices, receive data from them, perform various operations over data (e.g. aggregation, multidimensional analysis, transformation, etc.), and visualize it in a suitable way for business. *Azure IoT Suite* addresses the challenge of having a full-featured IoT framework as a combinations of three different sub-problems: scaling, telemetry patterns, and big data. *Azure IoT* supports a wide range of hardware devices, operating systems, and programming languages.

1) Architecture: A high level overview of *Azure IoT's* architecture is provided in Figure 7 [54]. IoT devices interact with *Azure cloud* through a predefined cloud gateway. The incoming data from these devices is either stored in the cloud for further processing and analytics by Azure cloud services (e.g. Azure Machine Learning and Azure Stream Analytics) or offered immediately to some services for real-time analytics. The output of both tracks is presented and visualized in a customized way that fits the desires of customers and suites their business.

Azure IoT Hub [55] is a web service that enables bi-directional communication between devices and the cloud backend services taking into account all security requirements. The cloud sends messages to devices in terms of either commands or notifications. Commands are orders to devices to perform actions, whereas notifications are information needed in some cases during the lifecycle of the execution of some commands. For

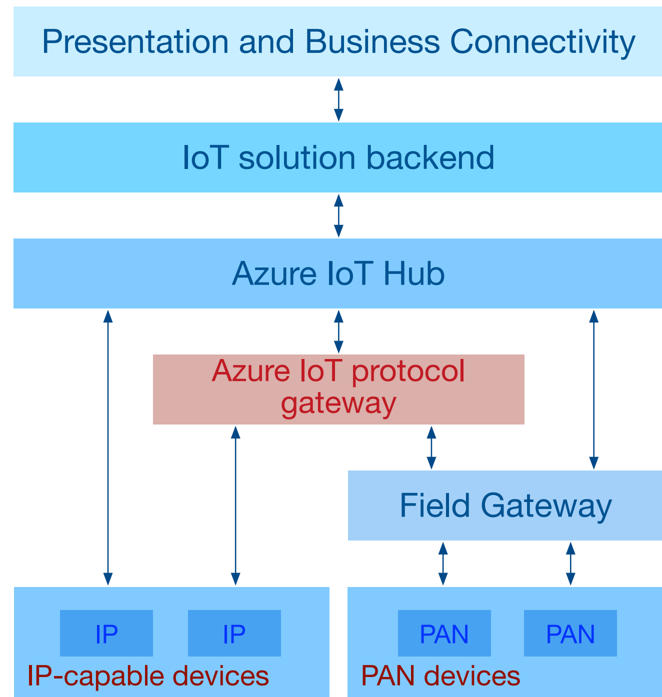

Fig. 7: Azure IoT Architecture

each command being sent, the cloud backend should receive a feedback from the device as a confirmation message of successful delivery, or a delivery fault message to warn about the delivery failure status. Similarly, devices send messages to the cloud backend in two formats: telemetry data or commands outcome. *Azure IoT hub* has an *identity registry* for holding the identity and authentication related information of each device. Also, it has *device identity management* unit to manage all connected and authenticated devices.

There are two classes of IoT devices: IP-capable and PAN. IP-capable devices have the ability to communicate with Azure IoT Hub directly by implementing one of the supported communication protocols [56]. *Azure IoT Hub* natively supports communication over AMQPs, MQTT or HTTP protocols. Support for additional protocols is possible via *Azure IoT protocol gateway* [57]. The gateway allows for protocol adaptation. Some devices and field gateways might not able to use one of the supported protocols by *Azure IoT Hub*. In this case, they can communicate with *Azure IoT Hub* via *Azure IoT protocol gateway* which acts as a bidirectional bridge. It reduces the gap between the different communication protocols, and tries to find a common language between all involved parties. From one side, the protocol gateway uses MQTT/AMQP protocol to communicate with *Azure IoT Hub* directly. From the other side, it is adaptable to support a variety of communication protocols depending on the connected device standards.

The Field Gateway is simply an aggregation point for PAN (personal area network) devices. Since these constrained devices do not have enough capacity to run secured HTTP sessions, they send their data to the field gateway to aggregate, store, and forward it securely to *Azure IoT Hub*.

The *IoT solution backend* layer represents a wide range of

Azure cloud services [58] (e.g. Azure Machine Learning, Azure Stream Analytics, etc.).

The top layer of *Azure IoT* architecture is the presentation layer. Users are free to visualize their data as they want. Microsoft provides the Business Intelligence (BI) service to present data in an effective and attractive way [59].

2) Smart Application Specifications: Microsoft provides various SDKs to support different IoT devices and platforms. IoT device SDKs along with IoT service SDKs are provided in order to make developers able to connect to *Azure IoT Hub* and let users manage their devices. The IoT device SDKs enable developers to implement client applications for a wide variety of devices ranging from simple network-connected sensors to a powerful standalone computing devices. Up to now, C, Node.js, Java, Python, and .NET programming languages are supported in such SDKs [60].

3) Hardware Specifications: Azure IoT supports a wide range of operating systems and hardware devices. The following conditions must be satisfied in each device in order to have the ability interact with *Azure IoT cloud* [60]:

- **TLS support**: for secure communication.
- **SHA-256 support**: for authentication purposes.
- **Memory footprint**: the memory footprint mainly depends on the SDK and the protocol used, in addition to the platform targeted (e.g. the minimum requirement of RAM used by C SDK is 64KB).
- **Real Time Clock**: having a real time clock or being able to connect to an NTP server is important for establishing TLS connections and generating secure tokens for authentication.

Only IP-capable devices can communicate directly with *Azure IoT Hub* (see Figure 7). Other low-power constrained devices are able to connect via a field gateway if they satisfy the aforementioned conditions.

Compatible operating systems and platforms include Windows, Android, Debian, mbed OS, Windows IoT Core, Arduino, TI-RTOS, and many others. A complete list of the compatible operating systems, platforms and hardware devices exists in the *Azure Certified for IoT device catalog*[6].

4) Security Features: Azure IoT takes the advantage of the security and privacy built into the Azure platform, along with *Security Development Lifecycle* (SDL)[7] [61] and *Operational Security Assurance* (OSA)[8] [62] processes for secure development and operation of all Microsoft softwares. In the architecture of *Azure IoT*, security is embedded into every layer and enforced in each component of the ecosystem. Figure 8 gives an overview of *Azure IoT* security architecture[9] [63].

[6]https://catalog.azureiotsuite.com/

[7]SDL is a software security assurance process that helps developers to address security requirements and build more secure software along with reducing development cost.

[8]OSA is a framework incorporates a variety of security capabilities including SDL.

[9]The Figure has been taken from Microsoft Azure documentation.

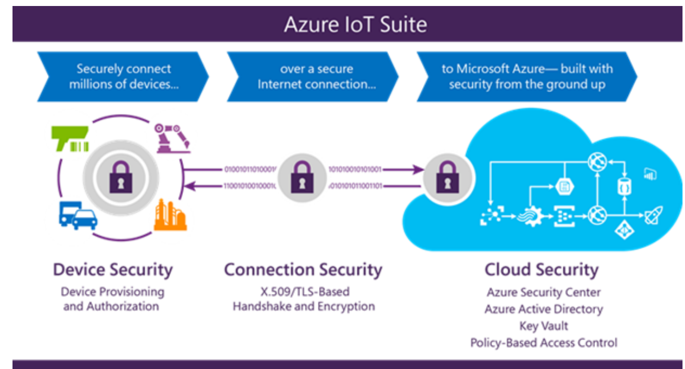

Fig. 8: Azure IoT Security Architecture

- **Authentication:** In order to establish a connection between IoT devices and *Azure IoT Hub*, mutual authentication is required. *Transport Layer Security* (TLS) protocol is used to encrypt the handshaking process. The cloud service is authenticated by sending an identity proof in terms of X.509 certificate to the targeted IoT device. *Azure IoT* issues a unique device identity key for each device at deployment time. The device then authenticates itself to *Azure IoT Hub* by sending a token contains an HMAC-SHA256 signature string which is a combination of the generated key along with a user-selected device Id.
- **Authorization and Access Control:** *Azure IoT* takes benefits of *Azure Active Directory* (AAD) [64] to provide a policy-based authorization model for data stored in the cloud, enabling easy access, management, and auditing. This model also enables near-instant revocation of access to data stored in the cloud, and of connected IoT devices. *Azure IoT Hub* identifies a set of access control rules to grant or deny permissions to either IoT devices or smart apps. System-level authorization makes access credentials and permissions near-instantly revocable. Therefore, The access control policies include activation and dis-activation of the identity of any IoT device.
- **Secure Communication:** SSL/TLS protocol is used to encrypt communication and ensure the integrity and confidentiality of data. The *identity registry* in *Azure IoT Hub* provides a secure storage of the identities of devices and security keys. Furthermore, data is stored in either DocumentDB [65] or in SQL databases, ensuring a high level of privacy.

D. Brillo/Weave

Google released *Brillo/Weave* platform for the rapid implementation of IoT applications. The platform consists of two main backbones: *Brillo* [66] and *Weave* [67]. *Brillo* [10] is an android-based operating system for the development of embedded low power devices, whereas *Weave* acts as a communication shell for interactions and message-passing purposes. The main role of *Weave* is to register a device over the cloud and send/receive remote commands. Both components complement each other and together form the IoT framework. *Brillo/Weave*

[10]Recently, Google released a rebranded version, called *AndroidThings* but it still does not support Weave to create a fully featured IoT framework.

is mainly targeting smart homes and expanding to support general IoT devices.

1) Architecture: Figure 9 provides an overview of the architecture of *Brillo/Weave* framework, which includes two sub-architectures belonging to *Brillo* and *Weave* respectively.

Brillo is a light-weight embedded operating system based on Android stack and fully implemented in C/C++ programing languages. It does not support any Java framework or runtime.

The bottom layer represents the platform of IoT devices. The *Kernel* layer is located at the top of the *Hardware* layer. It is Linux based and it has the responsibility to provide basic architectural model for managing system resources, process scheduling, communication with external devices when needed and so on. Also, It provides drivers and libraries to control displays, cameras, power, WiFi, keypads, and many other resources over the physical device. However, no graphics or GNU libraries are supported. The *android HAL* (Hardware Abstraction layer) is a middleware, which bridges the gap between the hardware and the software. It allows android applications to communicate with hardware specific device drivers by handling system calls between the kernel and the top android-based layers. Not shown in the architecture, *Brillo* uses *Binder IPC* mechanism [68] to interact with the android system services from the application framework.

Moving upwards, the *OTA Updates* component [69] is a wireless service aims to install batches and update versions of software over the air. The underlying devices perform regular checks with OTA servers for updates. Also, OTA servers notify all connected devices once there are some new updates available. *Metrics* component collects usage data from devices in order to analyze and view it to understand the behavioral patters of users. Also, crash reports can be submitted to debug remote devices.

While *Brillo* represents the low level segment (OS) of this architecture, *Weave*[11] is the high level one. It is a communication suite of protocols and APIs that lets smart phones, IoT devices, and the cloud to communicate with each others. In addition, it provides services for authentication, discovery, provisioning, and interaction. Practically, *Weave* is following a JSON format. As mentioned before, *Weave* module is baked into the *Brillo* OS as a significant part of the top layer in Brillo's architecture. *Weave* adds a key feature to the user experience through the capability to connect to devices directly or via the cloud. This is achieved by exposing a common language between all Brillo-powered devices, which is *Weave*. Furthermore, *Weave* exists as a mobile SDK for smart phones and a cloud-based web services for the cloud. Mobile SDK runs on either Android or iOS phones in order to connect mobile apps to the Brillo-powered IoT devices. Once the connectivity gets established, mobile apps can use either the local APIs, if they are located in the same network, or the

[11]Google Weave is totally different from NEST WEAVE protocol.

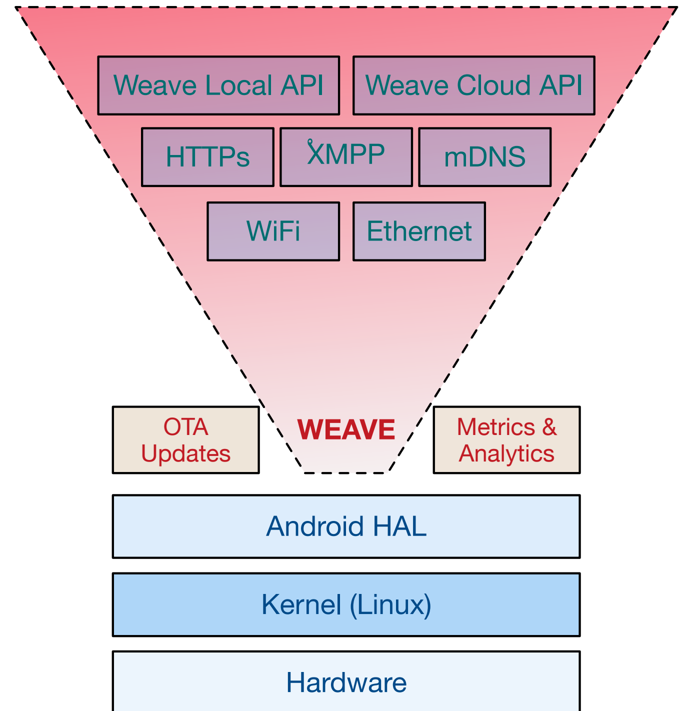

Fig. 9: Brillo/Weave Architecture

cloud APIs to control and manage the connected IoT devices. As shown in Figure 9, *Weave* supports multiple communication and application protocols.

To sum up, the underlying architecture illustrates the key building blocks of *Brillo/Weave* IoT framework. The last three layers represent the operating system, whereas the top layer includes the core services which composes of *OTA Updates*, *Weave*, and *Metrics and Analysis* services. Figuratively, the Brillo developer kit (BDK) is a necessary building block of the IoT platform [70] which is based on *Android.mk* build architecture. Using DBK, developers can perform local unit tests, integration tests, and build entire packages.

2) Smart Applications Specifications: Weave comes with a mobile SDK for both iOS and Android to build apps to control and enhance the connected device experience for mobile users. Any Android- or iOS-based mobile phone can run smart apps able to talk to Brillo-powered embedded devices. The smart app should include the *Weave* SDK as a communication module. In general, third party developers can implement applications in any platform using any programming language supports *Weave*. On the other side, IoT devices should run *Brillo* in order to interact with smart apps with no further requirements. Currently, only Google Cloud supports *Weave* and no other professional cloud (e.g. Amazon, Microsoft, etc.) does that.

3) Hardware Specifications: Brillo operating system is compatible only with Microprocessor (MPU)[12] devices that have a minimum memory footprint of at least 35 MB of RAM. ARM, Intel (X86), and MIPS are the only supported architec-

[12]For the difference between MPU and MCU devices, please refer to reference [71]

tures [70].

In particular, the minimum hardware requirements [72] [70] of the smart device to host *Brillo* are:

- 32MB RAM.
- 128MB ROM.
- support one of the following architectures: ARM, X86, or MIPS.
- WiFi 802.11n.
- Bluetooth 4.0+.

Commercially, the Intel Edison kit [73] with the Arduino expansion board is the first *Brillo* starter board.

4) Security Features: A high priority has been given for verifying security through out the design of both *Brillo* and *Weave*. Secure boot, signed over-the-air updates, timely patches at the OS level, and the use of SSL/TLS are all building blocks of the security architecture of *Brillo/Weave* framework.

- **Authentication:** *Weave* main functions is the Discovery, provisioning, and authenticating devices and users. OAuth 2.0 protocol along with digital certificates are used for authentication. Regardless the Weave-enabled cloud server chosen by the user, Google provides the authentication server.
- **Authorization and Access Control:** The right of access control is ensured by the Linux kernel. SELinux (Security Enhanced Linux) module is responsible for ensuring access control security policies, in which the owner of an IoT device can apply multiple levels of access control as needed. Enforcing access control is done by assigning the actual rights (read, execute, write) for each user or group of users.
  Again, as this IoT framework is Linux-based, sandboxing technique is applied with regards to UID (User Id) and GID (Group Id). It provides an enhanced mechanism to enforce the separation of information based on confidentiality and integrity requirements for each profile.
- **Secure Communication:** Secure communication are guaranteed via *Weave* by providing link-level security through the SSL/TLS protocol. Furthermore, the Linux kernel supports full disk encryption of saved data. Also, *Brillo* depends on a *Trusted Execution Environment* (TEE) and secure boot to protect code and data loaded inside the IoT and preserve confidentiality. The availability of TEE provides the connected devices *Hardware-backed keystore/ketmaster* [74].

E. Calvin

Calvin is an open source IoT platform released by Ericsson [75]. It is designed for building and managing distributed applications that enable devices talk to each others. *Calvin* is a framework that applies *Flow based Computing* (FBP)[13] paradigm [76] methodologies over the well-defined *actor model*[14] [77].

1) Architecture: Figure 10 shows the high level architecture of *Calvin*. The two bottom layers compose a foundation for the runtime environment[15]. The base layer represents the hardware or the physical device, whereas the second one encapsulates the operating system that the hardware exposes. At the top, the *platform dependent runtime* layer of *Calvin* takes a place. In this layer, all kinds of communications between different runtime environments (e.g. IoT devices) are handled. Also, this layer provides an abstraction of the hardware functionality (e.g. I/O operations). In other words, this layer supports several transport layer protocols (WiFi, BT, i2c) and presents the platform specific features like sensors and actuators in a uniform manner to the *platform independent runtime* layer where it resides above the *platform dependent runtime* layer. The *platform independent runtime* layer acts as an interface to the actors. The runtime can be configured to grant access to different resources depending if an actor is a part of the application or not. Actors execute asynchronously and autonomously per definition. They can also encapsulate protocols, such as REST or SQL queries, as well as device specific I/O functionality. Connections between actors are not specified in the architecture since they are logical and dynamically handled by the different runtimes.

Proxy Actors [78] is one of the important features that *Calvin* brings to the users. Using this attribute, Calvin-based applications can scale and function with non-Calvin ones. *Proxy Actors* help in integrating different systems as one system by handling communication and doing the task of converting data to messages or tokens that both systems can understand.

2) Smart Applications Specifications: Calvin framework divides the development process of an application into four pipelined isolated steps, each step has its own functionality as explained in the following [79]:

Describe: the functional part of any application which consists of reusable components or blocks called *Actors*. An actor is a component representing any object doing a computation e.g. smart phone, cloud, client, server, and etc. The way of communication between actors is by passing tokens over predefined ports. This is the only way to affect the behavior of an actor and change its state. Data is processed on the input ports of actors and then passed to the output ports in order to fire some actions depending on the contents of messages/tokens. Thus, writing an actor means identifying

[13]The FBP development approach views an application as a network of asynchronous processes communicating by passing messages as streams of structured data chunks, called information packets. This component-oriented model does not support single sequential processes which start at a particular point of time, do operations, and then finish to let others start their actions.

[14]The actor model is a mathematical theory that treats Actors as the universal primitives of concurrent digital computation. The model has been used as a framework for a theoretical understanding of concurrency.

[15]Runtime environment means the IoT device with the executable software loaded into it.

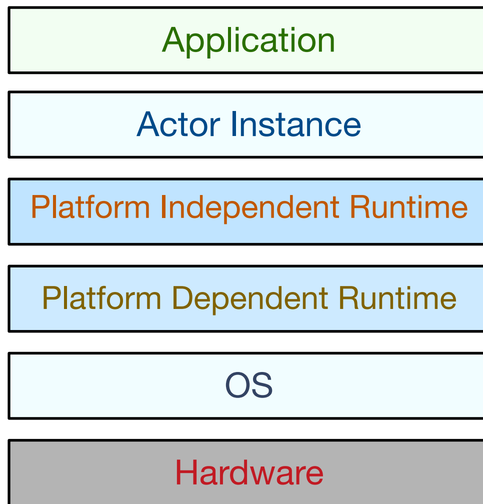

Fig. 10: Calvin Architecture

a new component that can be used in several locations by multiple applications. An actor can be created by (i) describing its actions, (ii) defining its input/output ports, (iii) identifying conditions for each particular action to be triggered, and (iv) adjusting the priority orders between actions.

Connect: in this interaction step, information about how actors are connected is supplied in a simple way using *CalvinScript*, a declarative language used to describe applications and how actors connected inside them. At the end of this phase, the application is completely identified and ready for deployment.

Deploy: after completing the two former steps, the deployment phase takes a place in order to run the application in reality. The core of this step is the lightweight distributed runtime that provides a number of accessible nodes for deployment and actors executions. Once the runtime environment is ready for execution after passing the application script to it, the distributed execution environment can move actors to any accessible runtime based on several factors such as resource, locality, connectivity, or performance requirement.

Manage: it monitors the life cycle of the application. Furthermore, it is involved in keeping track of the resource usage, firmware updates, error recovery, and scalability.

In order to support multiple programming languages and platforms, the design of *Calvin* does not require a specific way of processing data inside different actors. Only the format of data passed between ports is standardized. An API, written in python, is provided to device manufacturers and third party developers to port to *Calvin* runtime from various platforms and languages.

3) Hardware Specifications: *Calvin* framework supports different platforms, ranging from small sensor devices to data centers. Also, it is designed to run in distributed heterogeneous cloud environment. The only requirement needed in the hardware is the support of one of the compatible communication protocols.

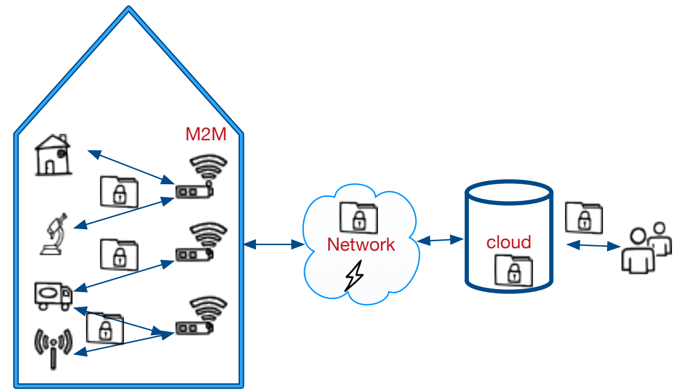

Fig. 11: Calvin Communication System

4) Security Features: *Calvin* platform applies security measures at different levels using various techniques [80].

- **Authentication:** Authenticating users can be done in three different ways. The first is via local authentication, in which the hash value of usernames and passwords are stored in a JSON file in a well-known directory in the same machine. Authentication can be verified by comparing the hash value of the entered and stored records. Second, using an external machine, which acts as an authentication server and performs the authentication on behalf of the corresponding runtime. Third, by using a RADIUS server. The radius server verifies the username and password and replies with subject attributes.
- **Authorization and Access Control:** Authorization is only supported via local or external procedure. In the local authorization, policies are stored in JSON files in a directory on the same machine, whereas the external authorization involves using another runtime to act as an authorization server. When external authorization is used, digital certificates in the form of X.509 standards are needed to verify signed JSON web tokens that contain the authorization request/response. The authorization process must be done after a successful authentication since it uses as an input the returned subject attributes. The access control is activated for a certain actor or entity via an attribute-based configuration file. Adding a feature with its value as an attribute means activating this feature in *Calvin* framework.
  To the best of our knowledge, neither sandboxing nor virtualization technique are provided in *Calvin* framework because Ericsson does not maintain their own cloud infrastructure.
- **Secure Communication:** Figure 11 shows an overview of the employed communication mechanism inside *Calvin* system. IoT devices can interact with each other or with smart applications. They are connected over short-range radio protocols to M2M gateways. Devices and gateways are integrated with the mobile network in order to access the cloud. End-users communicate with the cloud and explore the various information of the different IoT devices, that they authorized to access. IoT devices can not connect to the cloud via M2M gateways without conducting the authentication and authorization

processes. Since M2M gateways have no user interface for entering usernames and passwords, *Calvin* depends on the mobile networks and utilizes their capabilities. All M2M gateways are injected with SIM cards, and use their SIM-based identity to authenticate themselves to the cloud services using 3GPP standardized Generic Bootstrapping Architecture (GBA). The transmitted/received data may be secured using TLS/DTLS protocol. Elliptic Curve Cryptographic (ECC) algorithm is implemented as a part of the TLS suite and used for encrypting communications and providing digital signatures, as it incurs limited overhead, compared with other protocols (e.g. RSA). *Calvin* framework can be integrated with any public cloud system since it does not involve Ericsson cloud as a main component of the ecosystem. Therefore, *Calvin* does not provide details of the object level-security in the cloud.

F. HomeKit

HomeKit is an IoT framework released by Apple [81]. It is a platform dedicated only to home-connected IoT devices. It facilitates the process of managing and controlling connected accessories and appliances in a user's home by enabling interaction via smart apps. Through their own iOS devices, using the *HomeKit* app, called *Home*, users can discover, configure, control, and manage all *HomeKit* connected devices in a secure way. Furthermore, users can create actions and trigger their IoT devices using *Siri service* [82]. Until the moment of writing, iOS, watchOS, and tvOS are the only operating systems supporting the *HomeKit* capabilities.

1) Architecture: The core components of *HomeKit* architecture are: the *HomeKit* configuration database, *HomeKit* Accessory Protocol (HAP), *HomeKit* API, and the *HomeKit*-enabled devices.

Figure 12 simplifies the *HomeKit* architecture. The IoT devices (accessories) are located in the base layer. However, not all home-connected IoT devices can integrate with the *HomeKit* platform directly. They should meet some conditions as explained later in the hardware specifications section. Accessories that do not satisfy *HomeKit* requirements are still able to connect to the *HomeKit* platform using intermediate devices called *Bridges*. *HomeKit Bridges* are gateways that act as a proxy between iOS applications and home automations that do not support the *HomeKit* protocol. At the device side, the bridge supports only ZigBee and Z-Wave protocols. Therefore, the connected accessories are limited to support one of these protocols. For accessories that implements HAP, the bridge is not required and either IP (LAN, WiFi) or BLE is used as a transport protocol.

Since *HomeKit* speaks HAP, the backbone of the architecture is the HAP layer. HAP is proprietary protocol mapped over HTTPs with discovery leveraging the Bonjour architec-

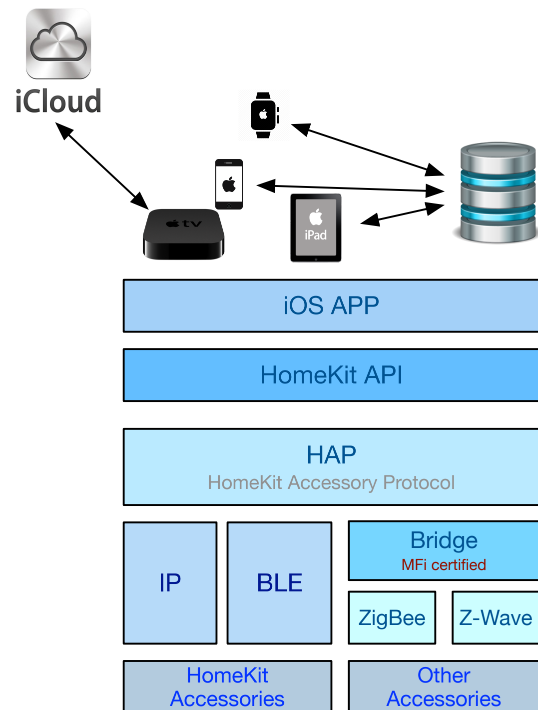

Fig. 12: HomeKit Architecture

ture[16] [83]. JSON format is used in HAP for exchanging messages between iOS apps and *HomeKit* compliant devices.

The *HomeKit API* layer is responsible for providing interfaces to third party developers to simplify the development of smart applications and hide the complexity of the underlying layers.

The application layer resides at the top of the architecture. It is responsible for providing a consistent user interface to all Apple devices sharing the same account, by synchronizing the stored data in the shared database using iCloud [84].

With tvOS 10 [85], Apple extended the capabilities on the Apple TV and *HomeKit* by bringing the *HomeKit* framework to the tvOS. Interestingly, Apple TV is able to run all home automations that users have set up inside their homes. Therefore, wherever users are, if they have an Internet connection, they can access their home accessories remotely. In other words, Apple TV acts as a hub or a gateway for home automations.

Apple TV also supports features for providing additional controls to shared users. This enables the possibility of any user to share the control of accessories with others, by inviting them using their Apple Id. It is also possible to grant administrative access to shared users. Shared users with an administrative access can change the configurations in the home. They can add or remove accessories as necessary. Also, they, in turn, can invite additional users to the home and let them control home accessories. Another possibility is controlling remote access per user. Using this functionality, the admin user can

[16]Bonjour is Apple framework for networking purposes. It implements a number of functionalities including: service discovery, address assignment, and hostname resolution.

grant or deny remote access capabilities to the other shared users.

2) Smart Application Specifications: An important part of the *HomeKit* ecosystem is the *Home* application. It is an Apple-designed app for *HomeKit* platform. It sets up home accessories as well as controls their common functions.

The *Home* app provides a very simple interface for users to set up, control, and configure accessories inside the home. The *Home* app is supported in all iOS devices and in the Apple watch. Using its user interface, the user can add a number of homes and define number of rooms in each home. Then, he start setting up and detecting accessories in each room.

Due to the integration with Apple system, *HomeKit* allows users to access their accessories when they are not at home. This remote access enabled through iOS device connectivity, in which the Apple TV acts as a gateway and intermediate layer between home-connected accessories and Internet-connected *Home* app or even third party apps.

Additionally, developers can implement iOS-based mobile applications and bring their apps to the foreground by utilizing the *HomeKit* API provided by Apple [86]. Using this API, developers can implement their applications by creating instances of a limited number of classes, delegating them to their apps and then customizing them according to the requirements. The architecture of *HomeKit* API is hierarchical. The entry point is the *Home Manager* class which provides pointers to a common database shared among all user's homes and maintains their data. Being shared, such database ensures consistency between all authorized applications in various devices. The *Home Manager* acts as a container of multiple homes and lets the user to label the primary one. Also, it lets the user to add or remove homes as necessary. Each home has to be uniquely identified and each one should points to its own data. Everything in a home must have a unique name as well. Narrowing the scope, the instance of the *Room* class lets users to add the number of rooms they have inside their homes. From a programming point of view, each room is an array of accessories. Each input of this array belongs to an instance of the *Accessory* class.

An accessory corresponds to the physical IoT device. Accessories are assigned and distributed between rooms. The instance of the accessory object allows users to access the device state. Also, accessories have to be uniquely named within a home. Names of accessories can be recognized by Siri service too. An accessory is the whole object that the a user is referring to. So, an accessory has a pointer back to the room where it is located, and it has a pointer to the array of services that represents its functionality. An accessory at any point of time may be reachable or not depending on the state of connectivity. This should be reflected in the smart application by maintaining the callback handler available to developers in the API [86].

Services represent the functionality of accessories. A service is described as a collection of characteristics. Characteristics are specific parameters that the user could interact with. Not all of services have names. The anonymous services are operational ones and not designed for user interaction (e.g. a firmware update service). Named services should be unique and exposed within the user interface. An example of such services are the light bulb and door bell. *HomeKit* does not only recognize names of services, but also takes into account Apple-defined service types. Therefore, users can refer to the service by its name or type when using Siri to detect it. The *Service* class contains the name of the service, an array of characteristics, service type, and a pointer back to the accessory. Characteristics provide some information and metadata about the state of the physical device. The characteristics can be of a few varieties: Read-only, Read-write, or Write-only. A good example is the thermostat device, where users want to read its temperature degree without writing privileges. This implies that the characteristics of this service should be Read-only [86].

HomeKit objects are stored in a database residing in the user's iOS device, which is synchronized over the iCloud to other involved iOS devices. This common database contains all information about homes and accessories that have been configured by users. It is available to all user's applications in a consistent way [86].

3) Hardware Specifications: HomeKit framework is compatible only with *HomeKit*-enabled devices. Thus, *HomeKit* supports all third-party hardware accessories that use Apple's MFi licensed technology [90] to connect electronically to the iPhone, iPad, iPod or Apple Watch. By using Apple's MFi license, Apple ensures that the produced hardware meets all key requirements and technical specifications of the *HomeKit* framework in terms of the supported communication protocols, physical security, etc.

As stated earlier, in order to connect an accessory, that is not MFi-certified, to the *HomeKit* framework, A *HomeKit* bridge must be used to find a common language between the heterogeneous transport protocols. The bridge supports only ZigBee and Z-Wave protocols from the input side of the accessory.

From a low level point of view, *HomeKit* supports a wide range of embedded microcontrollers including low-power, low-cost 32 bit MCUs. Both ARM and MIPS architectures are supported. Generally, the memory is the most critical resource in microcontrollers. However, for *HomeKit*, there is no minimum requirements for the size of memory since it mainly depends on the specific goal of the MCU and the size of the code loaded.

For users, *HomeKit*-enabled accessories can be controlled and managed only by Apple smart devices such as iPhones and iPads. There is no support for devices powered by other operating systems such as Android.

4) Security Features: HomeKit leverages many features from the security architecture of iOS [92] as it composes of software, hardware, and services designed to work together in a secure way, in which, end-to-end security must be guaranteed. This means that the entire ecosystem is covered by the security

polices and mechanisms enforced by the tight integration of hardware and software in iOS devices.

- **Authentication:** Authentication is required between *HomeKit*-connected accessories and iOS devices based on Ed25519[17] public-private key signature [93]. For each user and accessory in the *HomeKit* framework, an ed25519 key pair is generated for authentication purposes. Keys are stored in shielded keychain and synchronized between devices using iCloud Keychain. In the authentication process, keys are exchanged using Secure Remote Password protocol, in which a 8-digit code, provided by the accessory's manufacturer, must be entered by the user via the UI of the iOS device.
  Keys are encrypted using ChaCha20-Poly1305 AEAD with HKDF-SHA-512-derived keys [92]. The accessory's MFi certification is also verified during setup. The aforementioned keys are long-term keys. In order to protect each communication session, a temporary session key is generated using the Station-to-Station protocol and encrypted with HKDF-SHA-512 derived keys based on per-session Curve25519 keys [94]. The process of configuring Apple TV in order to perform remote access and the process of adding new shared users are also subjects to the same authentication and encryption mechanisms.
- **Authorization and Access Control:** Applications have to explicitly ask user's permissions to get access to their home data. Moreover, all applications are subject to security measures designed to prevent collisions and compromising each other. Sandboxing is enforced among apps. An application can access its own data only, which stored in a unique home directory. This directory is assigned randomly during the installation process of the application. On the other hand, iOS system data is isolated from third-party apps and users have no privilege to modify it in any case. Also, *Address Space Layout Randomization* (ASLR) technique [95] is applied to prevent buffer overflow memory-based attacks.
- **Secure Communication:** The integration of the core components of the iOS security architecture (e.g. secure boot, etc.) ensures that only trusted code can run in Apple devices. AES 256 encryption protocol is applied through an engine built into the DMA path between the flash storage and the main system memory in each device, making data encryption is highly efficient. Each Apple device has a unique device Id which is AES 256-bit key injected into the processor during manufacturing and this allows data to be cryptographically tied to one particular device only. This feature provides a robust secure hardware in case the memory chip is moved from a device to another one, the data is inaccessible and can not be read or decrypted. Apart from this, all cryptographic keys are created by the system's random number generator (RNG) using an algorithm based on CTR_DRBG [96].
  Communication using HTTP protocol are secured using TLS/DTLS with AES-128-GCM and SHA-256.
  In *HomeKit*, the long-term keys, used to secure communications, reside only in the user's devices. So even if the communication flows through an intermediate devices or services, the keys can not be decrypted even by Apple.
  Moreover, *HomeKit* provides *Perfect Forward Secrecy*, a property that ensures in every communication session between an Apple users's devices and their HomeKit enabled accessories, a new session key is generated for secrecy and confidentiality purposes. After the completion of the underlying session, this key is discarded. This feature strengthens the communication process in case, in the future, the device is compromised and the long-term key is publicly known, the adversary can not decrypt the communication process using only this long-term key.

G. Kura

Kura is an Eclipse IoT project which aims to provide a Java/OSGi-based[18] framework for IoT gateways that run M2M applications [98]. *Kura* offers a platform for managing the interaction between the local network of physical IoT devices and the public Internet or the cellular networks. Similarly to other frameworks, *Kura* abstracts and isolates the developer from the complexity of the hardware, networking sub-systems, and re-defining the development of existing software components, by offering an APIs that allow accessing and managing the underlying hardware smoothly [99].

1) Architecture: Figure 13 shows an overview about *Kura*'s architecture. *Kura* can only be installed on Linux-based devices and provides a remotely manageable system, complete with all the core services and a device abstraction layer for accessing the gateway's own hardware [100].

To interact with network-connected devices, smart applications can use Java's own networking capabilities to plug into the existing device infrastructure. The *device abstraction* layer allows developers to access many devices by abstracting the hardware using OSGi services for Serial, USB and Bluetooth communications. A communication API for devices attached via GPIO, I2C, or PWM will allow a system integrator to incorporate a custom hardware as a part of their gateway [101].

The *Gateway Basic Services* layer provides a configurable OSGi services available to applications to interact with the basic gateway functionality. Such services include watchdog, clock, GPS position, embedded database, process, and device profile service.

Also, the *network management* layer offers a configurable OSGi services to access the current network configuration and administer it (e.g. DHCP, NAT, DNS, etc.). It interacts with the

[17] https://ed25519.cr.yp.to/

[18] The OSGi specification Open Services Gateway initiative describes a modular system and a service platform that implements a dynamic component system for Java to simplify the process of developing reusable software building blocks. For more information, refer to [97].

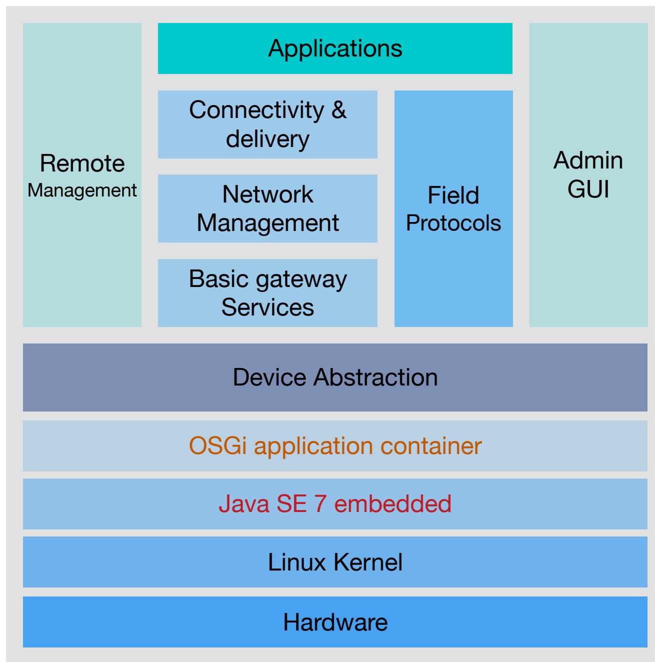

Fig. 13: Kura Architecture

Linux system to configure network interfaces including WiFi access points and PPP connections.

Furthermore, the *connectivity & delivery* layer simplifies the development of telemetry M2M applications interacting with a remote cloud server [102].

The functionality of *Remote Management* layer include remote configuration, remote software update, remote system command, remote log retrieval, device diagnostic service, and remote VPN access. Finally, The *administration GUI* provides interfaces for accessing such services.

2) Smart Applications Specifications: Java is the main programming language of *Kura* framework. An application is delivered as an OSGi module and run according to the standard specifications inside the container along with other components. The deployment of an application can be done remotely in the form of OSGi bundles. *Kura* package provides also a web front interface that allows developers to remotely login and manage their applications.

3) Hardware Specifications: Kura has two hard requirements in order to run on the IoT device. First, it must run at the top of Linux operating system. This means that the IoT device should be Linux-based. Second, Oracle Java VM 7 or later is required for *Kura* [103]. Memory size requirement depends on how large is the installed application and number of exchanged messages with other connected devices. An example of compatible devices, that meet the mentioned requirements, includes *Raspberry Pi* [104] and *BeagleBone* [105].

4) Security Features: The naive *Kura* framework provides a robust and simple security architecture for protecting and securing communications with IoT devices and gateways. However, there is a limited support for securely updating and configuring devices from cloud applications. To handle this issue, Eurotech [106] released an open-sourced ESF, a tool can be used along with *Kura* [107]. ESF adds support for advanced security, remote access via virtual private network (VPN), diagnostics and bundles for specific vertical applications. ESF maximizes the productivity by utilizing the basic *Kura* security API to make it easier to write Java applications that can ensure the integrity and security of new software bundles.

Eclipse foundation has injected also a number of security components into the *Kura* framework such as a security service, a certificate service, a secure sockets layer (SSL) manager, and a cryptography service.

- **Authentication:**
  Kura uses secure sockets provided by the Java Runtime environment. The Eclipse Paho client[19] [108] handles the majority of data communication via MQTT protocol [102]. This includes using public key cryptography to authenticate communication with remote devices and gateways.
- **Authorization and Access Control:** The security service component in *Kura* offers API to manage security policies and start script consistency, whereas the certificate service API is used to retrieve, store and verify certificates for SSL, device management and bundle signing. Ensuring the non-corruption or non-tampering with a file by a malicious user is done by doing a regular check of environmental integrity by the security manager component. ESF also enforces runtime policies to deny execution of particular services or the import/export of specific packages. This makes it harder for hackers to access the service for retrieving the master password from the device.
- **Secure Communication:** The SSL manager manages SSL certificates, trust stores and private and public keys. All communications are secured using SSL/TLS protocol. The cryptography APIs are used to encrypt and decrypt secrets and to retrieve the master password.

H. SmartThings

SmartThings is a platform released by Samsung for developing IoT applications. It is mainly dedicated to smart homes, where developers can implement applications that let users manage and control their home appliances via smart phones [109].

1) Architecture: According to Figure 14, the *SmartThings* ecosystem comprises of the following components: the *SmartThings cloud backend*, the *SmartThings hub/home controller*, the *SmartThings mobile client app* (the buddy app), and the IoT device (*SmartDevice*).

The hub (home controller) acts as a gateway between the IoT devices (SmartDevices) and the cloud services. It connects directly to the Internet and supports multiple communication protocols including ZigBee, Z-Wave, WiFi, and BLE. The

[19]The Paho Java Client is an MQTT client library written in Java for developing applications that run on the JVM or other Java compatible platforms such as Android.

SmartThings hub has the ability to execute some functionality locally without the need to connect to the cloud backend. Events are still required to be sent to the cloud once the hub gets online in order to reflect the current state of the home and execute other cloud-based services. Communication between all connected parties are encrypted using SSL/TLS protocol.

The buddy app, released by *SmartThings*, lets users access the home controller, manage their IoT devices smoothly, and, if required, install third party applications (SmartApps). The buddy app is supported by multiple mobile operating systems including Android and iOS. While the buddy app provides a basic and unified interface to all connected devices, SmartApps are customized applications, developed by third party developers, add more options and functionality to the end-user. Three classes of SmartApps are specified: (i) *Event-handlers*, (ii) *Solution Modules*, and (iii) *Service Managers*. *Event-handler* SmartApps allow end-users to subscribe to events and call handler methods upon their firings. *Solution Module* SmartApps act as a container for the two other categories of SmartApps and simplify the management of a certain physical area inside the home (e.g. bedroom). They are predefined by *SmartThings* developers and thus they can be installed via the *SmartThings* application interface (the buddy app). Lastly, *Service Manager* SmartApps are applications that integrate with SmartDevices and should be installed by end users in case of the presence of the device on the network. SmartApps may run on the hub as well as in the cloud depending on the physical characteristics of the SmartDevice.

SmartDevices may have the ability to connect via WiFi/IP protocol. This feature lets these devices to bypass the gateway and connect directly to the *SmartThings cloud*. Each SmartDevice belongs to one or more of the following categories: (i) *Hub-connected*, (ii) *LAN-connected*, and (iii) *Cloud-connected* [110]. *Hub-connected* devices include all devices that have the capability to interact with the *SmartThings* hub using ZigBee or Z-Wave home automation protocols, whereas *LAN-connected* devices have an extra feature which lets them to communicate with the hub over the LAN, e.g. *Sonos system*. *Cloud-connected* devices, e.g. *Ecobee thermostaat*, connect to the cloud directly using HTTP and authenticate themselves using *OAuth* protocol. Both LAN and Cloud-connected devices

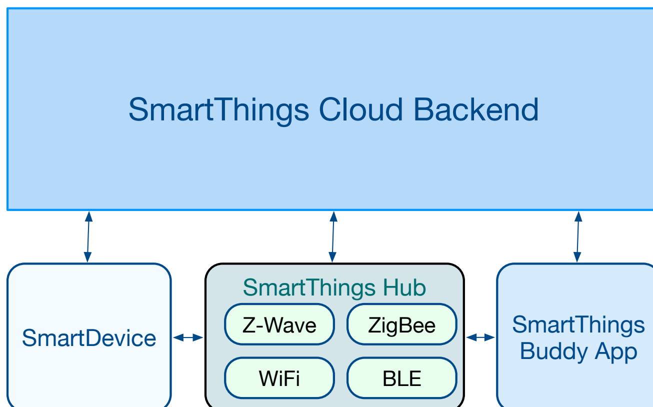

Fig. 14: SmartThings Architecture

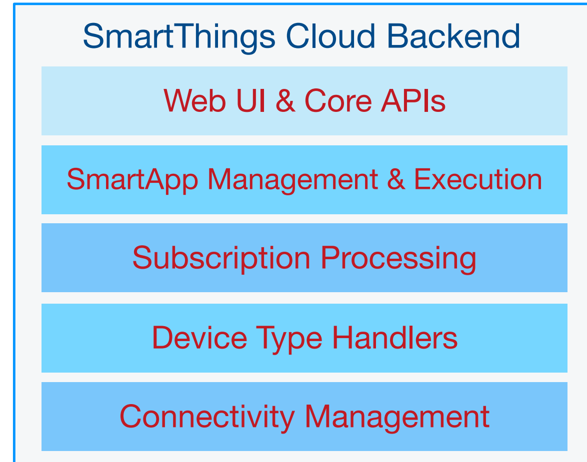

Fig. 15: The structure of the SmartThings cloud system

are able to communicate and integrate via web services like REST or SOAP [109].

There are two ways of communication between SmartApps and SmartDevices; (i) *Method calls*, in which, SmartApps can execute and perform operations over SmartDevices, and (ii) *Event-Subscription*, where SmartApps can subscribe to events generated by other SmartApps or SmartDevices.

Figure 15 gives an overview of the key building blocks of the *SmartThings cloud* [111]. The *Connectivity Management* layer is responsible for maintaining persistent and secure connection between the connected device (e.g. the hub) and cloud services. The *Device Type Handlers* layer simplifies the scalability by maintaining an instance or a virtual image for each type of SmartDevices. End-users interact with the physical SmartDevices indirectly via instances, hosted in the cloud. The *Subscription Processing* layer acts as an event manager for routing events from hubs/devices to SmartApps that are subscribed to a specific SmartDevice/event. The *SmartApp Management & Execution* layer provides access rights to the stored data, and is responsible for the execution of the SmartApp when triggered via either subscriptions or external calls. The top layer of the stack is the *Web UI* layer which provides web services and APIs in order to support the integration with third party applications.

The *SmartThings cloud backend* has two important functionality. First, it hosts and run SmartApps in a closed source environment. Second, it runs the virtual software image of the physical SmartDevice. In other words, it provides the abstraction and intelligence layers as well as web services that support the application layer.

2) Smart Applications Specifications: SmartApps should be implemented using a web-based IDE, offered by *SmartThings*, and in Groovy programming language [112]. Following a particular structure, a SmartApp is composed of five sections: *definition, preferences, predefined callbacks, event handlers*, and *mappings*. The latter is optional and only required for cloud-connected SmartApps. The *definition* section holds the metadata of the application (e.g. application name, author, etc.). The *preferences* section is responsible for defining the

TABLE I: Examples of Capabilities in SmartThings

Resource Name	Capability	Command	Attributes
Switch	capability.switch	on(), off()	switch(status: string)
Energy Meter	capability.energyMeter	-	energy(status: Integer)
Thermostat	capability.thermostat	off(), heat(), cool(), fanOn(), ...	temperature, thermostaatMode(status: string)
Smoke Detector	capability.smokeDetector	-	smoke(status: string) //possible values:detected, clear, or tested
Notification	capability.notification	deviceNotification(status: string)	-

target group of devices, specifying their capabilities and then reflecting the information to the user interface for interaction purposes. The *pre-defined callbacks* are methods already defined and automatically called upon meeting some conditions during the life cycle of the SmartApp. Finally, the *event handlers* section contains the handler methods of the various events.

3) Hardware Specification: SmartThings platform supports a wide variety of IoT devices that may either integrate with the *SmartThings* hub or connect directly to the cloud backend. These devices are manufactured by several vendors like Samsung, Google, Amazon, Philips Hue and many others. The only required specification is the ability to communicate using one of the compatible protocols.

4) Security Features: SmartThings has a security architecture that specifies what SmartDevices a SmartApp may access and what services can a SmartApp utilize in the authorized SmartDevice. In the following, we are discussing the security attributes verified by this architecture.

- **Authentication:** Integrating new SmartDevice in *SmartThings* environment involves the use of *OAuth/OAuth2* protocol for authenticating this SmartDevice and authorizing *SmartThings* platform to access its capabilities. Cloud- and LAN-connected devices follow a bit different procedure for authentication due to the use of other communication protocols to bypass the gateway and connect directly to the cloud. Both of them require identifying a custom *service manager* SmartApp along with a *device handler* for establishing connections, handling authentication, granting authorization, and maintaining communication. The main functions of the *service manager* are handling authentication with 3rd party cloud service, device discovery, initiating connection using OAuth protocol, and controlling SmartDevice actions. The *device handler* is responsible for parsing messages being sent or received by the corresponding SmartDevice. On the other hand, identifying the SmartDevice through out the authentication process is based on many factors due to the wide range of the supported SmartDevices from various vendors. Examples of such factors include unique identifier e.g. serial number, media access control (MAC) address, unique IP address, and so on.
- **Authorization and Access Control:** Accessing SmartDevices using SmartApps follows the policies governed by the *SmartThings Capability model*. *Capabilities* is an important concept in the underlying architecture which belongs to a logical layer that provides an abstraction of the capabilities of SmartDevices. The SmartApp should ask for a permission to use a capability offered by a SmartDevice. The capability, as identified by its name, is composed of a set of commands and their associated attributes. Commands are methods or functions to perform some actions on the SmartDevice, whereas attributes are input parameters representing the state of the device. Table I provides some examples of some capabilities in the SmartThings platform. As a consequence of applying this model, installing *a battery-monitoring* SmartApp will be authorized to use the capability of battery and prevented from accessing other resources or capabilities supported by the SmartDevice.
  All SmartApps are executed by the *SmartThings* ecosystem. This means that these apps run either in the closed-source cloud or on the *SmartThings* hub. The *SmartThings* infrastructure environment applies Kohsuke sandboxing technique [113] and isolates both SmartApps and SmartDevices (Device Handler instances) from each other [87]. In the sense of providing a highly controlled environment by Groovy, Kohsuke sandbox is an efficient implementation that isolates untrusted running pieces of code and allows only method calls that are predefined in a white list, stored in the restricted operating system. Developers can not create their own classes or load external libraries in such environment and once they publish a SmartApp or a SmartDevice, a private isolated data store is assigned.
- **Secure Communication:** The *SmartThings* Hub is a security-enabled Z-Wave product. When a security-enabled Z-Wave device is added to the Hub's network, communication will be encrypted using 128-bit AES. As the hub also supports the ZigBee protocol, it provides the same security guarantees for ZigBee-enabled products. In general, communications between all building blocks of the *SmartThings* ecosystem is performed over a SSL/TLS protocol.

V. Discussion

The IoT is where the Internet meets the physical world, in which, a completely new dimensions to security should be investigated as the attack threat moves from manipulating information to controlling actuations. The frameworks, included in this survey, approach IoT from the perspectives and priorities of their vendors. At the hardware level, there is a gap between these frameworks in terms of compatibility. This issue is due to the requirements and dependencies of the other components of the ecosystem of each framework (e.g. OS, security requirements). For example, IETF Class-1 IoT devices can be integrated with *AWS IoT* framework, and they are not

supported in *Brillo/Weave* because they do not have sufficient memory to allocate the operating system. *HomeKit* connects only to IoT devices that meet the minimum level of security by supporting Apple's MFi licensed technology. At the software level, some frameworks support any programming language for apps development (e.g. *AWS IoT*), whereas some others are limited to specific programming languages (e.g. *SmartThings* supports only Groovy). At the security level, each framework encapsulates its own security logic and applies the model that implements this logic. However, they follow the same trend and enforce the same security standards in some aspects. For example, for securing communications between IoT elements, they all use SSL/TLS protocol. For access control, they behave a little bit differently; some of them implements sandboxing techniques and some others propose their own models (e.g. capability model in *SmartThings*, configuration files in *Calvin*, etc.). Various cryptography primitives and cipher suites are supported by each framework depending on the availability of either supported software libraries or hardware modules. Techniques used to perform the mutual authentication between the involved parties in each framework are limited to the coverage domain and the capabilities of communication protocols. Theoretically, the presented security architectures seem to be robust and immune against potential threats. However, design flaws still exposes users to significant security risks if good practices in both design and implementation are not followed. *Fernandes et al.* [87] constructed four proof-of-concept attacks by exploiting two design flaws in *SmartThings* framework. On the other hand, some security challenges are still not handled by the majority of IoT frameworks. The vast majority of IoT devices depend on the commercial of the shelf (COTS) microcontrollers, and these devices are deployed without hardware security support. However, the design of the security models of the current frameworks does not consider these devices. Encryption techniques need higher computing power than what the simplest type of IoT devices can provide. Some frameworks (e.g. *HomeKit*) create and inject the secret key of the IoT device prior deployment to be used for the whole lifetime of the device. This key can't be changed after deployment. This increases the overall on-boarding time and threatens the privacy as, generally, IoT entities may not be owned by a single user (e.g. selling or exchanging this device between multiple users). Moreover, the embedded device may outlive the encryption algorithm lifetime, causing a cavity in the security architecture. For example, smart meters could last beyond 40 years, whereas crypto algorithms have a limited lifetime before they are broken. Therefore, they need to be updated frequently. Physical protection is still another security challenge couldn't be handled easily in IoT frameworks. Deployed IoT devices can be stolen or moved from their locations. This requires a physical protection of the IoT device to secure sensitive information in its memory. Addressing the privacy of the outlined frameworks was challenging due to the lack of information in some of them. Privacy should be ensured in all levels of the architectures. SDKs offered to third party developers to implement their IoT apps should preserve privacy in terms of preventing generating traceable signatures of the location and behavior of the individuals by applications. Finally, the flexibility of the security framework is a requirement. For example, If a cloud server is undergoing a Denial of Service (DoS) attack, the secure availability of data for end-users should be verified by outsourcing it from a secondary server. For a critical industrial processes that rely on time, the availability of data is of paramount importance. This feature is not ensured by frameworks such as *Kura* as it is M2M framework and does not offer its own cloud system. The user of *Kura* has to handle it himself by choosing a cloud server that meet this property.

Table II presents a comparison of the characteristics of each IoT framework.

VI. Conclusion

The IoT market is growing rapidly and as a consequence the attention has shifted from proposing single IoT elements and protocols towards application platforms in order to identify frameworks supporting the standard IoT suites of regulations and protocols. This study has covered a subset of commercially available frameworks and platforms for developing industrial and consumer based IoT applications. The selected frameworks have the same design philosophy in terms of identifying cloud-based applications by centralizing distributed data sources. However, they followed various approaches in order to apply this philosophy. A comparative analysis of the frameworks was conducted based on the architecture, hardware compatibility, software requirements, and security. We highlighted on the security measures of each framework as verifying the various security features and immunity against attacks is one of the most important contemporary issues facing the Internet of Things.

VII. Acknowledgment

This research is supported by the research fund of KU Leuven and iMec, a research institute founded by the Flemish government.

TABLE II: A brief summary of the characteristics of IoT frameworks

IoT Framework	SmartThings	AWS IoT	Calvin	Brillo/Weave	Kura	ARM Mbed	HomeKit	Azure IoT
Company	Samsung	Amazon	Ericsson	Google	Eclipse	ARM	Apple	Microsoft
Architecture Components	+ Cloud Backend + Smart Devices + SmartThings Hub + SmartThings Home App.	+ Cloud services + Smart devices + Device Gateway + Rules Engine + Registry Unit + Device Shadow	+ Actors: smart embedded devices, smart phones, cloud, servers. + Flow based computing	+ physical devices with Brillo/Android as OS + OTA servers + Cloud Services	Java/OSGi based.	+ Mbed OS + Mbed device Connector + mbed Cloud + mbed Client + ARM Cortex-M MCU	+ Home Conf. D.B. + HAP + HomeKit API + HomeKit-enabled devices	+ Cloud backend + Cloud Services + Cloud Gateway + Smart Devices
Programming Language	Groovy	Any language can use Restful API	+ CalvinScript + Python + others	Any programming language can talk through Weave SDK	Java	+ C++ for device side + multiple for user side	+ Swift + Objective-C	+ C + Node.js + Java + Python + .Net
Hardware Dependencies	+ SmartThings Hub	+ (optionally) AWS hub	NONE	NONE	NONE	+ ARM MCU	+ (optionally) Apple TV + (optionally) HomeKit bridge	+ Azure IoT Hub
Software Dependencies	The Home app.	NONE	NONE	+ Brillo OS + Weave SDK	+ JVM 7.0 or later	+ mbed OS + mbed Client	+ iOS + watchOS + tvOS + HomeKit app.	NONE
Compatible Hardware	All MCUs that support compatible communication protocols.	Any MCU can be configured using C, arduino platforms, or Node.js	Any MCU with communication capabilities	Any MCU with memory ¿= 35 MB	Linux based devices that support JVM 7.0+	+ 32 bits ARM Cortex-M MCUs	+ All devices that support Apple's MFi licensed technology + All devices can connect to HomeKit bridge	All devices that have 64KB RAM and RTC and support SHA-256
Supported Application Protocols	+ HTTP	+ HTTP + WebSockets + MQTT	+ HTTP	+ HTTP + XMPP	+ MQTT + CoAP	+ CoAP + HTTP + MQTT + others	+ HTTP	+ HTTP + MQTT + AMQP
Supported Communication Protocols	+ ZigBee + Z-wave + WiFi + BLE	All	+ WiFi + i2c + BT + others	+ WiFi + BLE + Ethernet	+ WiFi + BLE	All	+ WiFi + BLE + ZigBee + Z-wave	+ WiFi + ZigBee + Z-wave + others
Security – Authentication	+ OAuth/ OAuth2 protocol.	+ X.509 Certificates + AWS IAM + AWS Cognito	+ X.509 Certificates + Sim-based Identity	+ OAuth 2.0 + TEE	+ secure sockets	+ X.509 Certificates + other standards (mbed TLS)	+ Ed25519 public/private key signature + Curve25519 keys	+ X.509 certificates + HMAC-SHA256 signature
Security – Access Control	+ Capability mode/ Rules for granting permissions + Sandboxing Technique	+ IAM Roles + Rules Engine + Sandboxing	+ Configuration files	+ SELinux + ACL + Sandboxing: UID&GID	+ Security Manager + Runtime Policies	+ uVisor + MPU	+ Sandboxing + iOS security architecture + ASLR Technique	+ Azure Active Directory Policies + Access control rules of Azure IoT hub
Security – Communication	+ SSL/TLS	+ SSL/ TLS	+ SSL/ TLS	+ SSL/TLS	+ SSL/TLS	+mbed TLS	+ TLS/DTLS + Perfect Forward Secrecy	+ TLS/DTLS
Security – Cryptography	+ 128-bits AES protocol.	+ 128-bits AES + other primitives	+ ECC protocol	Full disk encryption supported by Linux kernel	Multiple cryptography primitives	+ mbed TLS + Hardware Crypto.	+ 256-bits AES + many others	Multiple cryptography primitives

REFERENCES

[1] D. Singh, G. Tripathi, and A. J. Jara, "A survey of internet-of-things: Future vision, architecture, challenges and services," in *Internet of things (WF-IoT), 2014 IEEE world forum on*, pp. 287–292, IEEE, 2014.

[2] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the internet of things," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–8, IEEE, 2015.

[3] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pp. 257–260, IEEE, 2012.

[4] Z. Specification, "Zigbee alliance," *URL: http://www. zigbee. org*, vol. 558, 2006.

[5] Z-Wave, "Z-wave public specification." http://z-wave.sigmadesigns.com/design-z-wave/z-wave-public-specification/. Online; accessed: April 2017.

[6] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, vol. 12, no. 9, pp. 11734–11753, 2012.

[7] A. Ghosh, R. Ratasuk, B. Mondal, N. Mangalvedhe, and T. Thomas, "Lte-advanced: next-generation wireless broadband technology [invited paper]," *IEEE Wireless Communications*, vol. 17, no. 3, pp. 10–22, 2010.

[8] E. Rescorla, "Http over tls," 2000.

[9] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," tech. rep., 2014.

[10] D. Locke, "Mq telemetry transport (mqtt) v3.1 protocol specification." http://www.ibm.com/developerworks/webservices/library/ws-mqtt/index.html. Online; accessed: April 2017.

[11] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," 2011.

[12] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, p. 87, 2006.

[13] O. M. Group, "Data distribution service v1.2." http://www.omg.org/spec/DDS/1.2/. Online; accessed: April 2017.

[14] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[15] Z. Sheng, S. Yang, Y. Yu, A. V. Vasilakos, J. A. McCann, and K. K. Leung, "A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities," *IEEE Wireless Communications*, vol. 20, no. 6, pp. 91–98, 2013.

[16] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, "A survey on security and privacy issues in internet-of-things," *IEEE Internet of Things Journal*, 2017.

[17] J. S. Kumar and D. R. Patel, "A survey on internet of things: Security and privacy issues," *International Journal of Computer Applications*, vol. 90, no. 11, 2014.

[18] B. Vikas, "Internet of things (iot): A survey on privacy issues and security," 2015.

[19] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized protocol stack for the internet of (important) things," *IEEE communications surveys & tutorials*, vol. 15, no. 3, pp. 1389–1406, 2013.

[20] T. Borgohain, U. Kumar, and S. Sanyal, "Survey of security and privacy issues of internet of things," *arXiv preprint arXiv:1501.02211*, 2015.

[21] I. Bouij-Pasquier, A. A. El Kalam, A. A. Ouahman, and M. De Montfort, "A security framework for internet of things," in *International Conference on Cryptology and Network Security*, pp. 19–31, Springer, 2015.

[22] P. Fremantle and P. Scott, "A survey of secure middleware for the internet of things," *PeerJ Computer Science*, vol. 3, p. e114, 2017.

[23] Amazon, "Aws iot framework." https://aws.amazon.com/iot. Online; accessed: April 2017.

[24] Amazon, "Amazon dynamodb." https://aws.amazon.com/dynamodb. Online; accessed: April 2017.

[25] Amazon, "Amazon s3." https://aws.amazon.com/s3. Online; accessed: April 2017.

[26] Amazon, "Amazon machine learning." https://aws.amazon.com/machine-learning. Online; accessed: April 2017.

[27] Amazon, "Components of aws iot framework." https://aws.amazon.com/iot/how-it-works/. Online; accessed: April 2017.

[28] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-sa publish/-subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*, pp. 791–798, IEEE, 2008.

[29] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2." https://www.ietf.org/rfc/rfc5246.txt. Online; accessed: April 2017.

[30] Amazon, "Amazon iot protocols." http://docs.aws.amazon.com/iot/latest/developerguide/protocols.html. Online; accessed: April 2017.

[31] Amazon, "Amazon lambda." https://aws.amazon.com/lambda. Online; accessed: April 2017.

[32] Amazon, "Amazon management console." https://aws.amazon.com/console. Online; accessed: April 2017.

[33] Amazon, "Amazon kinesis." https://aws.amazon.com/kinesis. Online; accessed: April 2017.

[34] Amazon, "Amazon command line interface." https://aws.amazon.com/cli. Online; accessed: April 2017.

[35] D. Cooper, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile." https://tools.ietf.org/html/rfc5280. Online; accessed: April 2017.

[36] Amazon, "Iam users, groups, and roles." http://docs.aws.amazon.com/iot/latest/developerguide/iam-users-groups-roles.html. Online; accessed: April 2017.

[37] Amazon, "Amazon cognito identities." http://docs.aws.amazon.com/iot/latest/developerguide/cognito-identities.html. Online; accessed: April 2017.

[38] Amazon, "X.509 certificates." http://docs.aws.amazon.com/iot/latest/developerguide/x509-certs.html. Online; accessed: April 2017.

[39] Amazon, "Aws identity and access management (iam)." https://aws.amazon.com/iam/. Online; accessed: April 2017.

[40] Amazon, "Amazon cognito." https://aws.amazon.com/cognito/. Online; accessed: April 2017.

[41] Amazon, "Signature version 4 signing process." http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html. Online; accessed: April 2017.

[42] Amazon, "Aws authorization." http://docs.aws.amazon.com/iot/latest/developerguide/authorization.html. Online; accessed: April 2017.

[43] ARM, "Arm mbed iot device platform." http://www.arm.com/products/iot-solutions/mbed-iot-device-platform. Online; accessed: April 2017.

[44] ARM, "mbed device connector." https://www.mbed.com/en/platform/cloud/mbed-device-connector-service/. Online; accessed: April 2017.

[45] ARM, "mbed os." https://www.mbed.com/en/platform/mbed-os/. Online; accessed: April 2017.

[46] A. mbed, "mbed client." https://www.mbed.com/en/platform/mbed-client/. Online; accessed: April 2017.

[47] A. mbed, "mbed device connector." https://docs.mbed.com/docs/getting-started-with-mbed-device-connector/en/latest/Connector-intro/. Online; accessed: April 2017.

[48] A. mbed, "mbed cloud." https://cloud.mbed.com/. Online; accessed: April 2017.

[49] A. mbed, "mbed documentation." https://docs.mbed.com/. Online; accessed: April 2017.

[50] A. mbed, "mbed security." https://www.mbed.com/en/technologies/security/. Online; accessed: April 2017.

[51] A. mbed, "mbed uvisor." https://www.mbed.com/en/technologies/security/uvisor/. Online; accessed: April 2017.

[52] A. mbed, "mbed tls." https://tls.mbed.org/core-features. Online; accessed: April 2017.

[53] Microsoft, "Tap into the internet of your things with azure iot suite." https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite. Online; accessed: April 2017.

[54] M. Azure, "Microsoft azure iot reference architecture." https://azure.microsoft.com/en-us/updates/microsoft-azure-iot-reference-architecture-available/. Online; accessed: April 2017.

[55] M. Azure, "Azure iot hub." https://azure.microsoft.com/en-us/services/iot-hub/. Online; accessed: April 2017.

[56] M. Azure, "Communication protocols." https://azure.microsoft.com/en-us/documentation/articles/iot-hub-devguide-messaging/#communication-protocols. Online; accessed: April 2017.

[57] M. Azure, "Azure iot protocol gateway." https://azure.microsoft.com/en-us/documentation/articles/iot-hub-protocol-gateway/. Online; accessed: April 2017.

[58] M. Azure, "Azure products." https://azure.microsoft.com/services/. Online; accessed: April 2017.

[59] Microsoft, "Power bi." https://powerbi.microsoft.com. Online; accessed: April 2017.

[60] M. Azure, "Azore iot sdks." https://github.com/Azure/azure-iot-sdks. Online; accessed: April 2017.

[61] Microsoft, "Security development lifecycle." https://www.microsoft.com/en-us/sdl/default.aspx. Online; accessed: April 2017.

[62] Microsoft, "Operational security assurance." https://www.microsoft.com/en-us/SDL/OperationalSecurityAssurance. Online; accessed: April 2017.

[63] M. Azure, "Internet of things security from the ground up." https://azure.microsoft.com/en-us/documentation/articles/iot-hub-security-ground-up/. Online; accessed: April 2017.

[64] M. Azure, "What is azure active directory." https://azure.microsoft.com/en-us/documentation/articles/active-directory-whatis/. Online; accessed: April 2017.

[65] M. Azure, "Documentdb." https://azure.microsoft.com/en-us/services/documentdb/. Online; accessed: April 2017.

[66] Google, "Brillo." https://developers.google.com/brillo/. Online; accessed: April 2017.

[67] Google, "Weave." https://developers.google.com/weave/. Online; accessed: April 2017.

[68] A. Gargenta, "Deep dive into android ipc/binder framework," in *AnDevCon: The Android Developer Conference*, 2012.

[69] Google, "Ota updates." https://source.android.com/devices/tech/ota/. Online; accessed: April 2017.

[70] J. MSV, "Google brillo vs. apple homekit: The battleground shifts to iot." http://www.forbes.com/sites/janakirammsv/2015/10/29/google-brillo-vs-apple-homekit-the-battleground-shifts-to-iot/#484c33674cac. Online; accessed: April 2017.

[71] F. Gaillard, "Microprocessor (mpu) or microcontroller (mcu)," *What factors should you consider when selecting the right processing device for your next design*, vol. 3, 2013.

[72] CNXSoft, "Brillo android based os for iot projects supports arm, intel and mips platforms." http://www.cnx-software.com/2015/10/28/brillo-android-based-os-for-iot-projects-support-arm-intel-and-mips-platforms/. Online; accessed: April 2017.

[73] Intel, "Getting started with brillo on the intel edison board." https://software.intel.com/en-us/articles/getting-started-with-brillo-on-the-intel-edison-board. Online; accessed: April 2017.

[74] Android, "Hardware-backed keystore." https://source.android.com/security/keystore. Online; accessed: April 2017.

[75] Ericsson, "Open source release of iot app environment calvin." https://www.ericsson.com/research-blog/cloud/open-source-calvin/. Online; accessed: April 2017.

[76] J. P. Morrison, *Flow-Based Programming, 2Nd Edition: A New Approach to Application Development*. Paramount, CA: CreateSpace, 2010.

[77] C. Hewitt, "Actor model of computation: scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010.

[78] Ericsson, "A closer look at calvin." https://www.ericsson.com/research-blog/cloud/closer-look-calvin/. Online; accessed: April 2017.

[79] P. Persson and O. Angelsmark, "Calvin merging cloud and iot," *Procedia Computer Science*, vol. 52, pp. 210 – 217, 2015.

[80] Ericsson, "Security in calvin." https://github.com/EricssonResearch/calvin-base/wiki/Security/. Online; accessed: April 2017.

[81] Apple, "The smart home just got smarter." http://www.apple.com/ios/home/. Online; accessed: April 2017.

[82] Apple, "Siri." http://www.apple.com/ios/siri/. Online; accessed: April 2017.

[83] Apple, "About bonjour." https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/NetServices/Introduction.html. Online; accessed: April 2017.

[84] Apple, "icloud." http://www.apple.com/lae/icloud/. Online; accessed: April 2017.

[85] Apple, "tvos." http://www.apple.com/tvos/. Online; accessed: April 2017.

[86] Apple, "Homekit developer guide." https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/HomeKitDeveloperGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40015050. Online; accessed: April 2017.

[87] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[88] Apple, "Homekit - api reference." https://developer.apple.com/reference/homekit. Online; accessed: April 2017.

[89] Apple, "Xcode." https://developer.apple.com/xcode/. Online; accessed: April 2017.

[90] Apple, "Mfi program." https://developer.apple.com/programs/mfi/. Online; accessed: April 2017.

[91] OberonHAP, "Setting the benchmark for homekit implementations." https://oberonhap.com/. Online; accessed: April 2017.

[92] Apple, "ios security." http://www.apple.com/business/docs/iOS_Security_Guide.pdf. Online; accessed: April 2017.

[93] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[94] D. Bernstein, "A state-of-the-art diffie-hellman function." https://cr.yp.to/ecdh.html. Online; accessed: April 2017.

[95] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 574–588, IEEE, 2013.

[96] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators." http://dx.doi.org/10.6028/NIST.SP.800-90Ar1. Online; accessed: April 2017.

[97] O. Alliance, "Osgi architecture." https://www.osgi.org/developer/architecture/. Online; accessed: April 2017.

[98] E. Organization, "Kura framework." http://www.eclipse.org/kura/. Online; accessed: April 2017.

[99] E. Organization, "Kura framework." http://wiki.eclipse.org/Kura. Online; accessed: April 2017.

[100] E. Organization, "Kura - osgi-based application framework for m2m service gateways." http://www.eclipse.org/proposals/technology.kura/. Online; accessed: April 2017.

[101] E. Organization, "Kura - a gateway for the internet of things." http://www.eclipse.org/community/eclipse_newsletter/2014/february/article3.php. Online; accessed: April 2017.

[102] E. Organization, "Mqtt and coap, iot protocols." http://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php. Online; accessed: April 2017.

[103] E. Organization, "Kura - hardware targets." http://eclipse.github.io/kura/ref/hardware-targets.html. Online; accessed: April 2017.

[104] E. Organization, "Kura - raspberry pi quick start." http://eclipse.github.io/kura/doc/raspberry-pi-quick-start.html. Online; accessed: April 2017.

[105] E. Organization, "Kura - beaglebone quick start." http://eclipse.github.io/kura/doc/beaglebone-quick-start.html. Online; accessed: April 2017.

[106] Eurotech, "Eurotech." https://www.eurotech.com/en/about+eurotech/. Online; accessed: April 2017.

[107] G. Lawton, "How to put configurable security in effect for an iot gateway." http://www.theserverside.com/tip/How-to-put-configurable-security-in-effect-for-an-IoT-gateway. Online; accessed: April 2017.

[108] E. Organization, "Eclipse paho." http://www.eclipse.org/paho/. Online; accessed: April 2017.

[109] SmartThings, "Smartthings documentation." http://docs.smartthings.com/en/latest/. Online; accessed: April 2017.

[110] SmartThings, "Cloud and lan-connected devices." http://docs.smartthings.com/en/latest/cloud-and-lan-connected-device-types-developers-guide/. Online; accessed: April 2017.

[111] SmartThings, "Smartthings architecture." http://docs.smartthings.com/en/latest/architecture/index.html. Online; accessed: April 2017.

[112] Groovy, "Groovy programming language." http://www.groovy-lang.org. Online; accessed: April 2017.

[113] K. Kawaguchi, "Groovy sandbox." http://groovy-sandbox.kohsuke.org/. Online; accessed: April 2017.