Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Autonomous Certificate Management for Microservices in Smart Spaces

Lorenzo Donini

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Autonomous Certificate Management for Microservices in Smart Spaces

Autonomes Zertifikatsmanagement für Microservices in Smart Spaces

Author:	Lorenzo Donini
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Dr.-Ing. Marc-Oliver Pahl
	M. Sc. Stefan Liebald
Date:	August 15, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, August 15, 2018	
Location, Date	Signature

Abstract

The Internet of Things (IoT) has been at the center of many emerging new technologies in the recent years. Interconnected smart devices constitute an essential part of the IoT paradigm. These become more and more powerful every year. In the near future, many common objects in smart spaces will be capable of running more complex software. The integration of IoT devices in standard internet infrastructures has, however, introduced numerous security challenges. The heterogeneity of devices and their constrained resources are the main cause for a lack of standards and security features. The implications of lacking security mechanisms include, among other, data integrity and confidentiality violations, unauthorized access to sensitive resources and arbitrary malicious code execution.

The Distributed Smart Space Orchestration System (DS2OS) is a framework built for managing smart spaces of the future, focusing on user experience and automated application lifecycle control on unattended nodes. It considers an app-based ecosystem, in which users can download applications from an app store and deploy them on their IoT devices, distributed across the smart space.

This work investigates a suitable approach for protecting the application lifecycle management in such a distributed scenario. The designed solution relies on the Virtual State Layer (VSL) middleware for securely exchanging data between different IoT nodes within the network. At the same time, it introduces a farsighted approach to securing apps targeting IoT devices. Apps are protected at different stages thanks to code-signing and certificate-based authentication. The major contribution is introduced with an automated certificate renewal mechanism, aiming to guarantee security and consistency of access control rules within a distributed system, where connectivity may be intermittent. In the end, the security against known attacks on the IoT is evaluated, along with considerations about the performance impact and the scalability of the designed solution.

Zusammenfassung

Das Internet of Things (IoT) stand in den letzten Jahren im Mittelpunkt vieler neuer Technologien. Verbundene intelligente Geräte sind ein essentieller Teil dieses IoT Paradigmas. Diese Geräte werden jedes Jahr immer leistungsfähiger. In der nahen Zukunft werden viele übliche “Dinge“ in Smart Spaces dazu fähig sein, komplexere Software auszuführen. Die Integration der IoT Geräte in die allgemeine Internetinfrastruktur bringt allerdings zahlreiche Sicherheitherausforderungen mit sich. Die Heterogenität der Geräte und deren eingeschränkte Ressourcen waren der Hauptgrund für einen Mangel an Standards und Sicherheitsfunktionen. Die fehlende Sicherheitsmechanismen haben zu unerwünschten Nebenwirkungen geführt, wie die Verletzung der Datenintegrität und Datenvertraulichkeit, sowie nicht autorisierten Zugriff auf wichtige Ressourcen und die Ausführung von schädlichem Code geführt.

Das Distributed Smart Space Orchestration System (DS2OS) bietet ein Framework, um Smart Spaces der Zukunft zu verwalten. Im Fokus steht die leichte Bedienung und automatisierte Regelung des Anwendungs-Lebenszyklus auf unbewachten Knoten. Das System basiert auf einem Anwendungs-Ökosystem, wo Anwendungen über einen App Store von Benutzern heruntergeladen werden, und auf deren verteilten IoT Geräten installiert werden.

Diese Arbeit erforscht einen geeigneten Ansatz um die Verwaltung des Lebenszyklus der Anwendungen in so einem verteilten System zu schützen. Der Lösungsentwurf baut auf der Virtual State Layer (VSL) Middleware auf, um Daten zwischen verschiedenen IoT Knoten innerhalb des Netzwerks sicher auszutauschen. Gleichzeitig wird ein weitsichtiger Ansatz vorgestellt, um Apps auf IoT Geräten zu sichern. Die Apps sind in jeder Phase des Anwendungs-Lebenszyklus mit Code-Signierung und Zertifikat-basierter Authentifizierung geschützt. Der Hauptbeitrag dieser Arbeit ist ein automatisierter Zertifikatserneuerungs-Mechanismus, welcher die Sicherheit und die Konsistenz der Zugriffsregeln in einem verteilen System mit unzuverlässiger Konnektivität gewährleistet. Abschliessend wird die Widerstandsfähigkeit gegen bekannte Angriffe auf das IoT analysiert, wobei auch die Auswirkungen auf die Leistung und die Skalierbarkeit der vorgeschlagenen Lösung bewertet werden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) is a global network of *things*, which are real-world physical objects with augmented capabilities. Things have embedded electronics, which allow them to perform complex operations and communicate with other devices.

During the past years the IoT has rapidly evolved without the appropriate consideration of the profound security goals and challenges involved.
Security in the IoT world is a complex topic, as most devices today have limited resources. To improve performance and battery life, many of these devices prioritize functionality over security. The plethora of heterogeneous devices available on the market runs mostly closed-source software, typically driven by economical decisions instead of reliability. This opens the way to numerous combinations of security issues and makes it difficult to provide proper protection.

Contrary to smartphones, modern IoT devices are typically left unattended by the user. In a scenario with billions of distributed nodes, unattended devices become a huge security concern and require proper protection. It must therefore be possible to manage the applications running on these nodes.

The research presented in [59] addresses this problem and aims to create an autonomous smart space service management. This assumption leads to a scenario in which applications can be installed/uninstalled on IoT nodes easily, via an app store. In an app-store-like scenario, it suddenly becomes even more important to focus on security. The management of the lifecycle of software and applications within the smart space is achieved using the Distributed Smart Space Orchestration System (DS2OS).

1.1 Topic and Goals

The topic tackled within this thesis is the IoT application security in distributed smart spaces. The thesis is strictly related to other two ongoing works [58] [12], which focus on providing service management mechanisms within the DS2OS. All three works are based on the research introduced in [59].

The concrete goals include:

1. Conducting an extensive literature review of the existing security threats and countermeasures for app-based and IoT scenarios
2. Improving the status quo of the DS2OS with additional security mechanisms aimed to protect the service management
3. Implementing a prototype which incorporates the designed security features
4. Evaluating the performance impact of the introduced security features

1.2 Outline

Chapter 2 will present a thorough analysis of smart spaces and the scenario targeted by the DS2OS. Furthermore existing IoT security issues and possible countermeasures, related to the chosen scenario, will be investigated.
Existing works, related to the analyzed topic, are presented in chapter 3. The approaches to security in related works will be compared to the proposed security concept.
Chapter 4 will cover the design of the concrete solution to the analyzed problems, matching the system requirements. Additional implementation details will be documented in chapter 5.
Chapter 6 contains an evaluation of the implemented prototype with respect to the overall system.
In the end, the results of the thesis are summarized and possibilities for future works are presented.

Chapter 2

Background and Analysis

A first analysis of smart spaces, and how users interact with them is necessary, to put them into context. The main focus will be put on the problem scenario and on the intrinsic issues and security limitations of smart spaces. These problems will be addressed thoroughly and possible solution techniques analyzed.

Some background on the topic and motivation will be given in 2.1, then the problem domain will be analysed.
Some history and considerations about smart spaces will be presented in 2.2.
Architectural patterns fit for IoT services and similar distributed systems in general are described in 2.3.
Section 2.4 introduces an existing approach that covers some of the issues identified before, leading to the guiding research questions.
Given the problem scenario, section 2.5 will identify and prioritize security goals for the IoT. Sections 2.6, 2.7 and 2.8 will then focus on security-relevant aspects for this thesis and possible approaches.

2.1 Motivation

IoT applications have different scopes and can target different scenarios and environments. The main applications nowadays are:

- *Smart buildings*: offices and homes equipped with smart devices enable energy management, dynamic lighting, door locks and so on.

- *Connected Health*: small devices with embedded sensors can become health monitoring and drug delivery systems [84], leading to improved well-being of people.
- *Wearables*: devices such as smartwatches, fitness bands, rings [55], smart shoes [16] and so on aid users in keeping track of their health data.
- *Environmental*: sensor nodes for monitoring the environment (e. g. air/water quality) collect enormous amounts of data, which allow to detect emergency situations and provide important insight to recurring problems.
- *Smart cities*: traffic optimizations, waste management, pollution monitoring, urban security and so on make life in cities easier and more efficient.
- *Connected vehicles*: moving IoT systems can interact with other environmental sensor and nodes to distribute information, improve traffic management, navigation and more.
- *Industrial*: tracking assets, production progress, detecting faults and so on greatly aid the management and control of production and assembly lines.
- *Energy management*: proper monitoring and prediction of energy consumption allows to improve the efficiency, reliability and costs of electricity. Smart grids provide macro-management, while users micro-manage their own power consumption at the same time.

Other emerging sectors, where connected smart devices make a difference are, for example, the food supply chain, farming, supply transportation and retail management.

It is obvious how the market evolved in just a few years and how more use-cases for smart devices are presented every year.

The chart in 2.1 shows the increase in installed IoT devices over the past three years. A prediction for the next 7 years is also given [74]. According to this prediction, by the end of 2025, around 75 billion IoT devices will be installed throughout the world. This means that the number of intelligent devices will have exceeded the global population by a factor of 10. Also, the amount of IoT devices alone will be dramatically superior to the amount of traditional computing devices, such as desktop computers.
This exponential increase in smart devices available on the market has produced an undesired side effect: each device is a potential threat that needs to be addressed. If traditional Information Technology (IT) security is critical, the security of the devices used in the above mentioned scenarios is no different.

On one hand, IoT devices are often vulnerable nodes, given their low computational resources and therefore lack of a proper security stack. On the other hand, companies

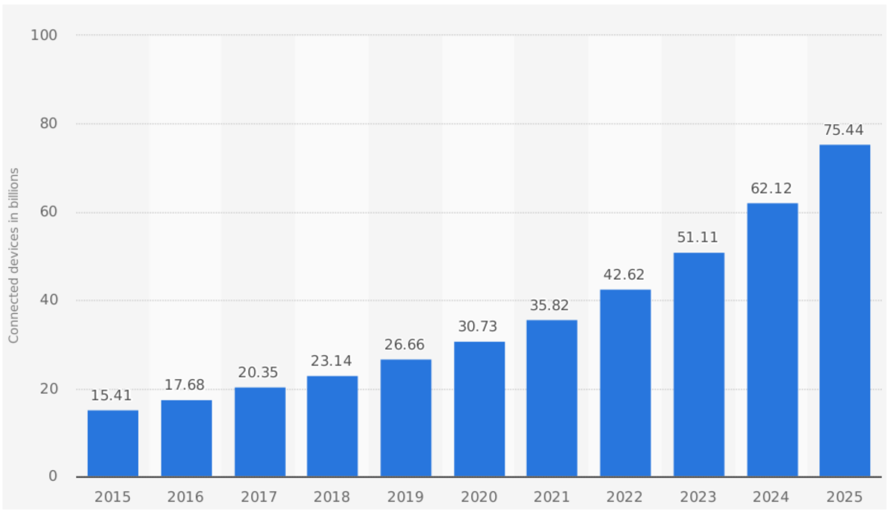


Figure 2.1: Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). Source: https://ihsmarkit.com/index.html

designing products to release on the market often overlook security implications and rather focus on the needed functionality. At the beginning of the IoT era, the primary requirements were to permeate the market with innovative and ground-breaking solutions. Being farsighted enough to consider all possible security issues that could arise was not a primary requirement. Nowadays, this has changed, given the plethora of devices and different purposes they serve.

Considering we are surrounded with smart devices with important responsibilities we have to consider what would happen in the face of malicious attackers. In short, the advent of pervasive computing, as integrating part of our lives, also opens up countless potential new security issues. These issues need to be identified and defended against.

Several major vulnerabilities in IoT devices have been recorded up until now [24]. Many more exist and are being discovered every day [41]. There is a multitude of reasons for attackers to take advantage of vulnerabilities: access to confidential data, profit, protests against organizations, wreak havoc and so on.
Regardless of the motivation of attackers, security vulnerabilities pose a real threat, to us and to our environment. For example, an implantable cardiac device may fail to recognize a heart problem or even administer fatal shocks, endangering a person's life. The same reasoning applies to many other health monitoring applications.
Other scenarios are not to underestimate either. Imagine a smart city paralyzed by an attack on the infrastructure itself: huge traffic congestions, large-scale electricity grid issues and improper functioning of the safety/security measures.

Another huge issue to be considered is the theft of sensible data. It represents a huge privacy violation and can have severe repercussions on individual's lives and maybe even the community as a whole. Gaining control over nodes can lead to attacks on third-party entities.

All the mentioned threats become very real as we rely more on machines and automate everything. It is fundamental to address these threats aggressively, not only by researchers but also by industry leaders.

2.2 Smart Spaces

The term Smart Spaces refers to environments containing several smart devices, such as a home, an office, a factory, and so on. Smart devices include any kind of device, powered by electricity and capable of running software. One defining characteristic of these devices, to which they owe the name *Smart device*, is the capability of communicating (often wirelessly) with other devices, in order to exchange information. This fundamental functionality broadens the scope of the single devices, allowing them to interact with any other device and access many more features.
In reality this is often not the case, given the heterogeneity of such devices and their incompatibility with one other. These incompatibilities are due to different hardware constraints, different protocols, vendor policies and so on.

2.2.1 Smart Home

The most common example of a modern smart space is the smart home. The high-level setting shown in 2.2 is a usual representation of a smart home.

Most of the deployed devices are typically sold by different vendors and each serve a single specific purpose. The heterogeneous nature of the deployed devices has a major side-effect:

<P.0> An ad-hoc service needs to be used in order to manage a single device. As a result most devices require a different service or app to manage it

For example retrieving data from a smart scale requires the user to connect to it over Bluetooth Low Energy (BLE), using a smartphone app built by the vendor for that specific purpose. Similarly, to set up smart lighting requires the user to have an app compatible with that specific light bulb. If light bulbs from two different vendors are deployed, different apps are needed.

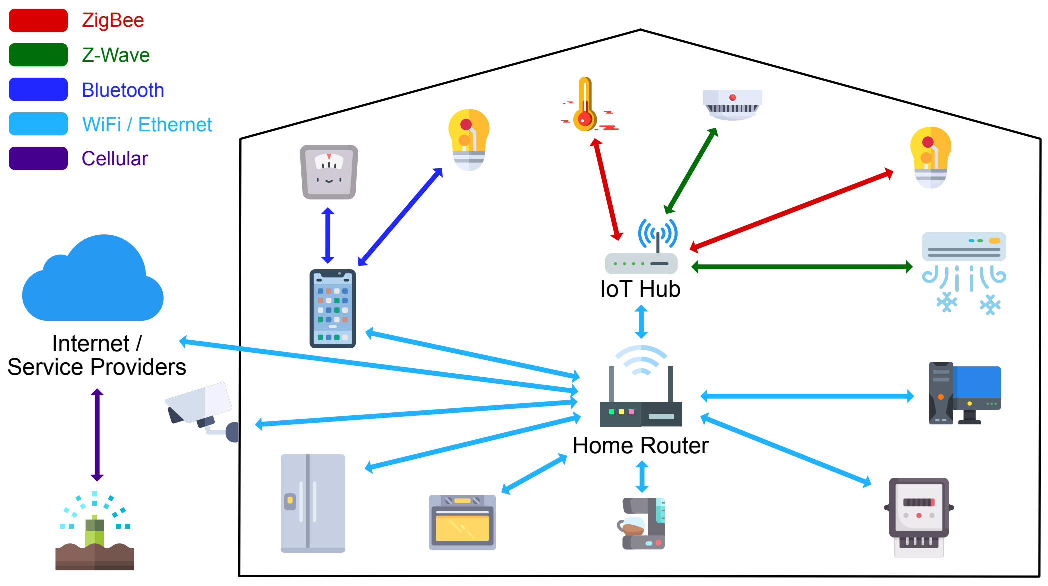


Figure 2.2: An example of a smart home containing heterogeneous IoT devices, which use different connectivity technologies

Depending on the hardware and the vendor, the same kind of device may also use different communication technologies. Regardless of the underlying connection, multiple application protocols are also possible.

Today, there is a plethora of communication technologies and application protocols used in smart spaces (not just smart homes). Some IoT devices use Constrained Application Protocol (COAP) on top of user datagram protocol (UDP). More powerful devices rely on the widely known, yet with more resource overhead, Representational state transfer (REST) over HyperText Transfer Protocol (HTTP).
This makes it extremely difficult to integrate them in a single platform or management system. A more uniform abstraction is required to address this problem.

Different approaches to bridge the gap between different vendors and software stacks exist. Some solutions found in literature and in the industry, both for consumers and businesses, are briefly introduced in section 3.1.

2.2.2 Hardware

IoT devices are all powered by micro-controllers or even more performant single-board computers (e.g. a Raspberry Pi [25]). Although constrained, the equipped resources are quite generous and not necessarily saturated by everyday use. As technology evolves and the demand for smart devices increases, computing units also get more powerful.

In the future it will be possible to have devices with increased capabilities for the same price (or even lower), maintaining the same footprint and power consumption. This is a key point, as it will allow to run multiple applications and more complex software on smart devices spread throughout a site.

Several projects exist, with the purpose of shrinking the Linux kernel to a minimum and make it deployable on very constrained devices and even micro-controllers [82]. Running an Operating System (OS) opens up many new possibilities and comes with security features that have been tested and used for many years.
When talking about Micro-Controller Unit (MCU), the minimum CPU, Random Access Memory (RAM) and Read-Only Memory (ROM) requirements are extremely low. An STM32F4 MCU [75] sports 180Mhz CPU, 384KB RAM and 2MB ROM. While this is indeed very low, it leaves little room for applications and doesn't contain many modules useful to the developers, such as a Memory Management Unit (MMU), cryptographic modules and more.
Other Single-Board Computer (SBC)s with a very small footprint exist, capable of running a full-fledged version of Linux. Worthy of mentioning are:

- Omega2+ [14], supported by a 580Mhz CPU, 128MB RAM and 32MB of storage
- VoCore2 [76], supported by a 580 CPU, 128MB RAM and only 16MB of storage
- C.H.I.P. [13], with a generous 1Ghz CPU, 512MB RAM and 4GB storage, given its small size
- Raspberry Pi Zero [26], also assisted by a 1Ghz CPU and 512MB RAM

The mentioned hardware configurations certainly look low for today's desktops standards, but a modern Linux kernel can still run on something as constrained as the mentioned SBCs. A mere 580Mhz CPU allows to run most of the existing desktop-class software. The actual performance will, of course, be impacted, but the functionality of the software is guaranteed. The average current draw of such boards is also just 200mA, for a power consumption of around 0.6W.

Such a feat was not possible 5 years ago. The advancement of technology has made it possible to build powerful yet very tiny computers. It appears extremely likely, that more resource will enable devices in future smart spaces to perform demanding edge computation, run machine learning algorithms and offer more functionalities.
A good comparison to this phenomenon is the mobile phone. Phones are the first devices to become very smart in a short span of time [56].

Twenty years ago, mobile phones were only able to send/receive calls and SMS. By the end of the first decade of the 21st century, mobile phones had already evolved to smartphones. These allwed users to easily accomplish more complex tasks, such as managing their calendar appointments, surfing the web, taking photographs and listening to MP3 music. Nowadays, smartphones have partially replaced personal computers, giving users all the computational power they need for daily tasks and beyond (e.g. machine learning), while being in a pocket-size form-factor.

An important feature of smartphones is that they run a full-fledged OS. Any modern OS has high security standards, that have been used and tested for many years.
Smartphones also embrace the App economy, supporting multiple *apps* to run on it. Most of these apps each have a specific purpose, but the user can control them all from the same device.
The very same concept is slowly applying to other devices surrounding us: watches, house-appliances, doors, switches and more. Most of these devices are not in our pockets at any given time, but they still are integrating part of our everyday lives. It seems reasonable that future IoT devices will all be powered by tiny yet powerful computers.

2.2.3 Distributed Smart Spaces

Similarly to the evolution of smartphones, IoT spaces also have the potential to embrace an app-based ecosystem. More powerful smart devices with more sensors and resources will be able to run multiple services.

The smart space scenario, however, is intrinsically different. Contrary to smartphone apps, IoT devices and services are *geographically distributed* and are not always reachable within a network. Users also require a mechanism and UI to easily install services on their IoT devices.

It is undesirable, from a user perspective, to have an ad-hoc management system for each IoT device. Furthermore it doesn't seem scalable, particularly in environments with many devices, to rely on a different app for every available device/service.

The main identified problems for the present and near future are summarized:

<P.1> distributed IoT devices need to be setup, managed and maintained

<P.2> distributed IoT nodes may be unattended and suffer from churn and unavailability

<P.3> managing services individually, via ad-hoc interfaces, is not a scalable approach

2.3 Microservices

This section analyzes an emerging architectural pattern, which is fit for IoT applications, due to their distributed nature.

2.3.1 Monolithic Architecture

Traditional monolithic applications tend to include many tightly coupled modules, which grow in size and complexity over time. This leads to development and scalability issues, suffering from “dependency hell“ and overwhelmingly large codebases. Eventually, this kind of applications become unmanageable.
An example of a monolithic architecture with high complexity is visible in Figure 2.3.

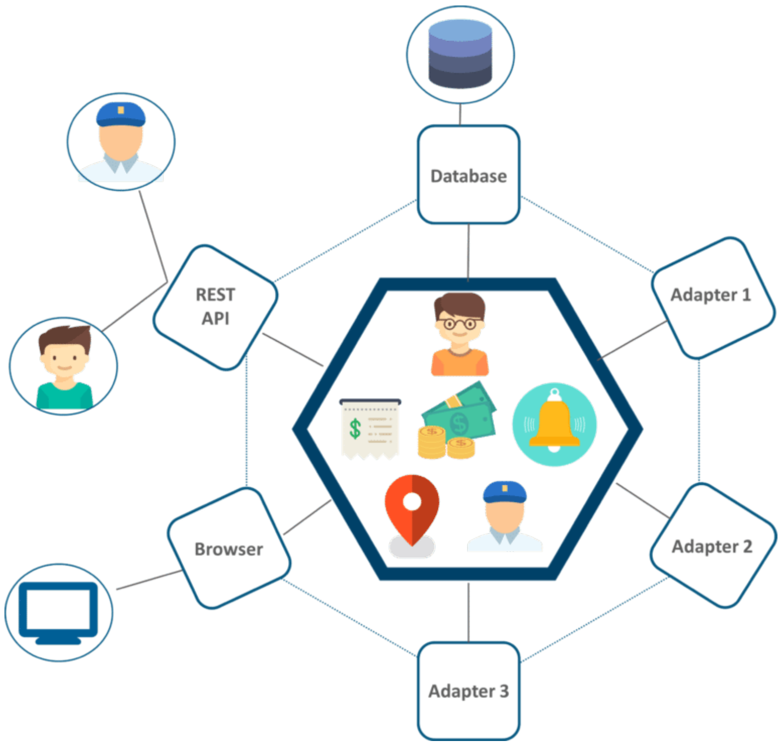


Figure 2.3: Monolithic architecture of Uber [44]

2.3.2 Microservice Architecture

The *microservice architecture pattern* [64] was invented to tackle the problems introduced by monolithic applications. Its architectural style structures an application as a collection of loosely coupled services.

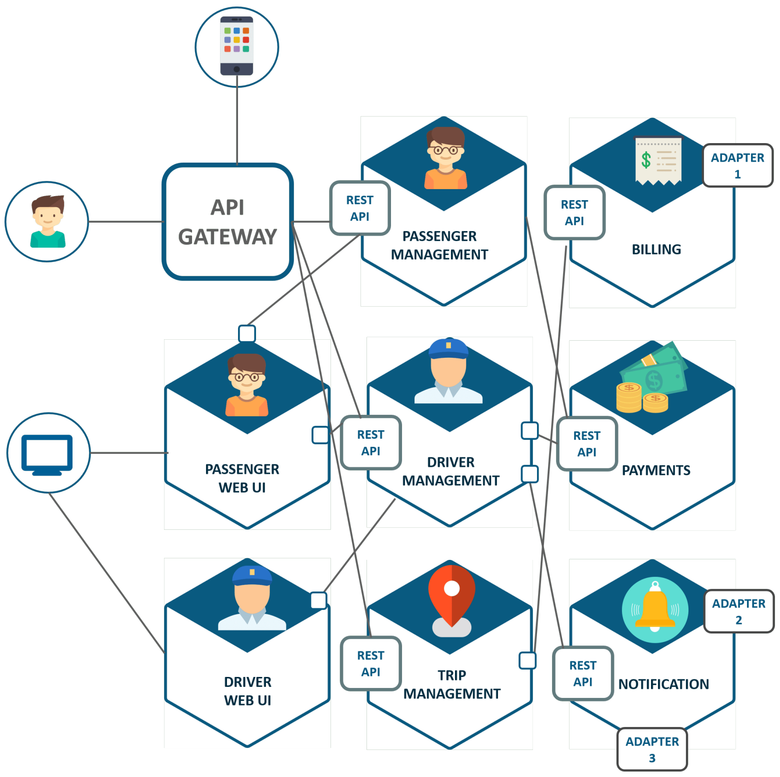


FIGURE 2.4: Microservice architecture of Uber [44]

Microservices architecture is a more concrete variant of the Service Oriented Architecture (SOA). A micro-service (μ-service) implements only a small subset of distinct and self-contained functionalities. Each service is a mini-application with its own hexagonal architecture, containing business logic and a database. It can expose Application Programming Interface (API) to other services and/or access other service's functionalities using adapters and interfaces. An example is shown in Figure 2.4.

Microservices typically communicate with one another using messages. Two kinds of messages exist:

- **Synchronous** messages, such as REST API;
- **Asynchronous** messages, such as Advanced Message Queuing Protocol (AMQP) [57] or Message Queuing Telemetry Transport (MQTT) [54]

Working with abstractions and technology-agnostic protocols enforces modularity and collaboration between services. Every μ-service is built with its own technology stack, independent of the technologies and frameworks used in other services (which it communicates with).

Such an architecture has several **advantages**:

- each μ-service is small, which makes it easier for developers to maintain;
- services boot faster as their size and complexity are rather small;
- the architecture enables continuous integration and delivery of large, complex applications, which is invaluable to organizations;
- small agile development teams can work on a single μ-service, without needing to know how other services work;
- each service leverages its own dedicated database, which is best suited to its needs;
- services can easily be swapped out, maintaining the same interfaces;
- errors are isolated to a single service and do not impact the entire system.

A microservices application is intrinsically a distributed system, which also introduces some **disadvantages**:

- some tasks become more complex, such as integration testing or implementing the communication mechanisms;
- deploying a single service requires more complex pipelines and an improved level of automation, as the number of μ-services increases;
- implementing use-cases or changes that involve multiple services at once requires more coordination
- better hardware is required, as each μ-service runs in a separate environment (e.g. a Docker container), allocating more memory and consuming more disk space.

This approach is used by many large organizations, such as Amazon and Netflix, to achieve better scalability and be able to better maintain their software services. It is especially used when dealing with large web applications.

2.3.3 IoT Model

Before comparing the IoT with the microservices approach described in 2.3.2, it is important to introduce the IoT Service Oriented Architecture (SOA).

The industry and research communities refer to the IoT reference model, shown in 2.5 and introduced by *CISCO* [78]. This seven-layer model is meant to be a widely-accepted model and facilitates the analysis and design of functionalities on a layer-basis.
The introduced model follows a bottom up vertical approach to offer an application.

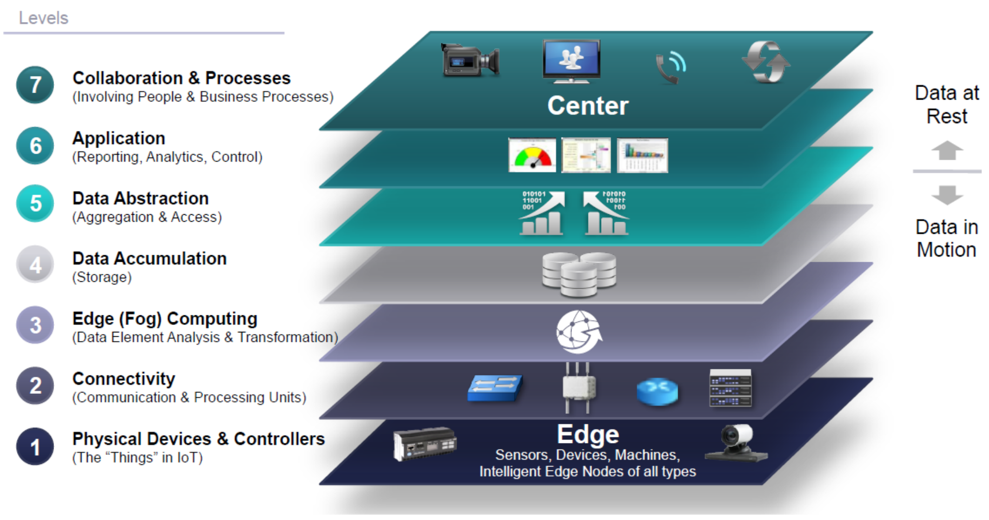

Levels
7 Collaboration & Processes
(Involving People & Business Processes)
6 Application
(Reporting, Analytics, Control)
5 Data Abstraction
(Aggregation & Access)
4 Data Accumulation
(Storage)
3 Edge (Fog) Computing
(Data Element Analysis & Transformation)
2 Connectivity
(Communication & Processing Units)
1 Physical Devices & Controllers
(The "Things" in IoT)
Center
Edge
Sensors, Devices, Machines,
Intelligent Edge Nodes of all types
Data at Rest
Data in Motion

FIGURE 2.5: The IoT reference model

In the IoT, each layer is independent from the others, following a modular approach (as seen in subsection 2.3.2). The resulting application is based on all layers working together to shape a complete service. The top-most layer may support collaboration between different IoT services, but this is often not the case nowadays, due to heterogeneity and vendor diversity.

2.3.4 MICROSERVICES FOR THE IoT

Modern smart spaces consider IoT nodes in a distributed environment, as introduced in subsection 2.2.3.
Self-contained IoT applications benefit from the μ-service architectural pattern the same way traditional services do. IoT applications can be considered as small and independent services, as they align with the capabilities and constraints of the embedded devices they represent. These services are built over several layers and typically consider:

- a hardware device, containing sensors, actors and storage;
- an on-device service, which acts as the Hardware Abstraction Layer (HAL) and provides the data;
- a back-end service, run on a more powerful machine without constraints;
- in some cases a front-end UI.

Many IoT top-level services nowadays run on cloud platforms. Cloud platforms support internet-based functions, such as maintenance, analytics, data storage, security and more. Exemplary platforms for cloud management that fulfill this purpose are Amazon Web Services [72], Google Cloud IoT [29], Azure IoT [51] and more.

Compared to traditional μ-service architecture, in IoT cloud μ-services, different vendors come into play, and getting services to collaborate is less trivial.
The vendor diversity leads to a major issue: every IoT service relies on a dedicated server and UI (be it a smartphone app or web application). In fact, the service decoupling principle described in subsection 2.3.2 is here taken to the extreme, as every service is truly independent.

There are several major documented differences when comparing “traditional“ μ-services to IoT services [11].

First, the main way for putting different services together in the IoT is by using an orchestrator, while in traditional μ-service architecture a choreography is best suited. The *orchestrator* is a centralized entity, in control of the other running services. In a *choreography*, every application is independent and is part of the resulting application.

Second, services can be containerized in both the μ-service architecture and the IoT. The most widely solution used for μ-services today is Docker [38]. It is a scalable approach which allows to package a μ-service and its dependencies in a virtualized container. Docker can be very heavyweight and taxing on nodes with hardware constraints. This makes it less suitable for IoT devices. Approaches more fit for IoT nodes are Docker versions with smaller footprint like RancherOS [39], or solutions like OSGi [2].

Last, in IoT scenarios it is considerably more difficult to perform continuous integration and delivery. This is mostly due to the problems mentioned in 2.2.3.
Continuous integration in the IoT is especially hard to achieve with services developed by different vendors, as they typically use custom interfaces and protocols. Service updates are common in microservices architecture, where approaches such as *blue-green deployment* [21] or *canary release* [68] are adopted, to reduce service downtime risks. Conversely, IoT nodes often do not have such requirements. Moreover, IoT services are typically unattended and therefore less maintained.

2.3.5 From Cloud to Edge Services

IoT services running on cloud platforms can be fully considered μ-services. The provided interface and functionalities are contained in the top-most layer, abstracting the underlying hardware.
Another valid approach, however, is to deploy IoT μ-services directly on edge nodes, completely off the cloud. Taking into consideration the increasing resources for edge

computation mentioned in subsection 2.2.2, it is completely feasible to deploy μ-services not just on cloud platforms but also on smart devices themselves. Edge nodes within smart spaces typically have internet connectivity and may be able to provide the same features offered by cloud platforms, possibly with comparable performance.

An important issue arises, when moving μ-services completely to the edge nodes:

<P.4> services deployed on unattended edge nodes require lifecycle management and additional security mechanisms

This problem is (partially) taken care of when cloud-based services manage the IoT. On distributed edge nodes, however, this problem needs to be addressed properly.

2.4 Distributed Smart Space Orchestration System (DS2OS)

The Distributed Smart Space Orchestration System (DS2OS) [59] is a high-level framework used for providing autonomous service management within a smart space. It is the result of a research project and is meant to bridge the gap between the existing IoT devices and the plethora of ad-hoc management systems built for them.

The DS2OS embodies the μ-service architecture described in 2.3, and pushes it towards an app-based ecosystem, targeting a fully distributed system.

The key idea of the DS2OS is to abstract the service management mechanisms and allow developers to develop their own μ-services without having to directly deal with any management-related issues, such as *service availability*, *Machine to machine (M2M) communication*, *network management*, *security* and so on.
Besides the benefits offered to the developer, the DS2OS also targets the final user and thrives to ease the management of a smart space. As described in 2.2, the smart space considered in this thesis consists of a plethora of nodes distributed in a site. These nodes are typically unattended and do not provide intuitive management interfaces themselves.
The average user doesn't want to spend much time configuring and managing every single node over its lifetime.

The DS2OS aims to solve problems [P.0], [P.1], [P.3] and [P.4] by introducing the idea of a central orchestrator node within a smart space, capable of managing all the others. By providing a single UI to the user, the entire smart space can be managed effortlessly and transparently.

The main features supported by the DS2OS to manage a smart space are:

- lifecycle management of μ-services
- resource usage collection on different nodes
- dynamic μ-service migration and load-balancing
- communication between nodes over a secure channel
- μ-service security (discussed in subsection 2.4.10)

By not allowing users to manually mess with the IoT devices, but only through a controlled management interface, the attack vector can be controlled and taken care of.

2.4.1 Architecture

The DS2OS is based on three components: the VSL, the S2Store and the μ-services running on the IoT nodes inside a smart space.

2.4.2 Virtual State Layer (VSL)

The Virtual State Layer (VSL) is a programming abstraction, based on a middleware concept introduced in [59].
The VSL is a novel attempt to gap the bridge between the cyber-world and the physical world. This is meant to overcome the heterogeneity of devices found in smart spaces. Each device in pervasive computing typically offers one or more services, tied to a specific context.
To unify the different contexts under a single interface, the VSL leverages a hybrid meta-model, used for defining *context models*.

Context Models

A context model provides an abstraction for the structure of the context of an IoT device. The context is represented using virtual objects, called context nodes. The mapping between a real-world entity and the context node representing it is purely abstract. This concept can, therefore, be applied to virtually any domain.

The meta-model introduced to describe a context allows to define:

- **regular** context nodes, which map a domain property to a specific value (e.g. a temperature value);
- **virtual** context nodes, which turn nodes into abstract service interfaces.

Both types of nodes serve as a representation of the underlying properties, but virtual nodes are not limited to just static data. They offer, instead, an interface for other services to retrieve dynamic data and trigger functionalities. This functionality is very powerful, as it allows to define virtual nodes as if they were high-level functions. This enables a SOA of smart space services.

Context models are defined using human-readable markup language, and support nested properties. The example in listing 1 shows the context model for a temperature service. The model contains a temperature value in Celsius and Fahrenheit, as well as some information about the temperature sensor.

```
<temperatureService type="basic/composed">
        <currentTemperature type="basic/composed">
                <celsius type="basic/number" />
                <fahrenheit type="basic/number" />
        </currentTemperature>
        <sensor type="basic/composed">
                <model type="basic/text" />
                <firmwareVersion type="basic/text" />
        </sensor>
</temperatureService>
```

Listing 1: Context Model defined in XML for a temperature service

The root element of a context model is the name of represented domain. All other nested context nodes are structured like a tree.
Each Context node offers primarily two functionalities: `get` and `set`. These are accessor functions that allow to respectively retrieve and update the data of a context node.
In the reference implementation of the VSL, set and get operations are invoked as if they were RESTful APIs. This holds true both for regular and virtual context nodes. Services also have the possibility to subscribe to a specific regular context node, to get notifications when the property data changes.

The VSL keeps a public Context Model Repository (CMR), in which the context models are uploaded. When a context model with a unique identifier is loaded by the VSL, it becomes usable within the smart space. Other services can access the data and functionalities offered by another service, via the context model.

Knowledge Repository

The VSL is built to support knowledge sharing and collaboration between services. For this purpose, it is designed as a fully distributed Peer-To-Peer (P2P) middleware system.

Each peer is called Knowledge Agent (KA). KAs manage the *Context Repository* and all the contained data.
The data contained within the Context Repository is structured like a tree, with the root of a tree being the KA itself. Every leaf of the root corresponds to a loaded context model. As already mentioned, each context is represented as a tree and contains *knowledge* about the modeled entity and its properties.

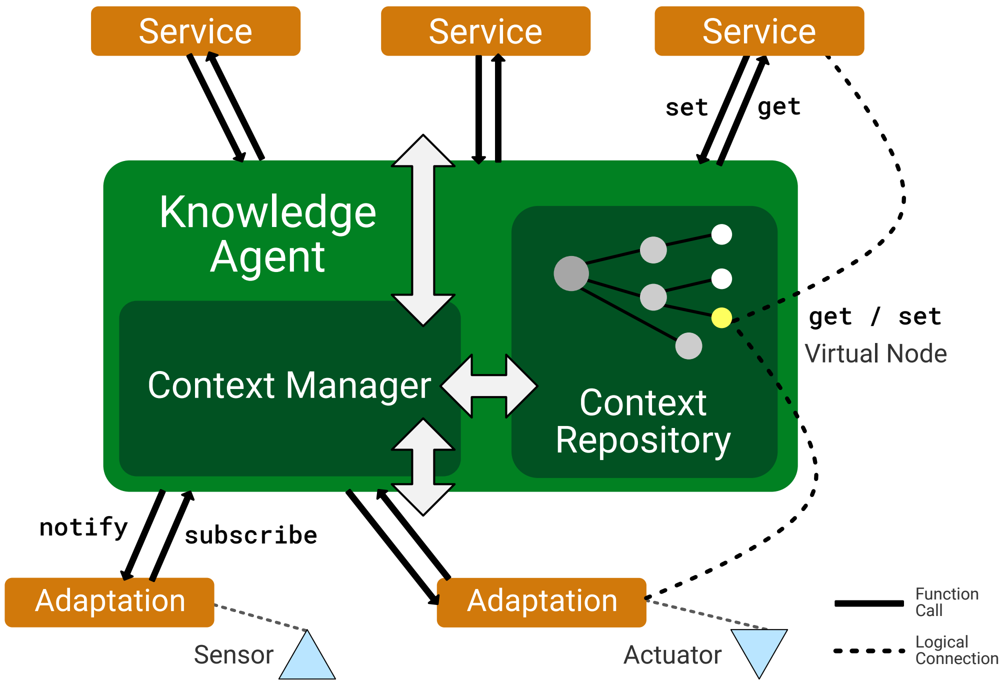


Figure 2.6: Architecture of Knowledge Agent (KA) with Context Management

Fig. 2.6 shows the components of a KA and its interaction mechanism with other services. Any kind of service conforming to the DS2OS framework can load their context model onto the VSL. Adaptation services are considered special services, as they don't just contain software logic but actually communicate with physical sensors and/or actuators.

As part of a distributed system, KAs provide context for services and retrieve context from other connected KA peers. The discussion about the synchronization procedure is not within the scope of this thesis, but more details about the lifecycle management and the current security concept are presented in subsection 2.4.6 and 2.4.10.

2.4.3 Smart Space Store (S2Store)

The S2Store store is the global entity in charge of storing μ-services, developer metadata, data analytics and more. Its functionality is conceptually the same as any other app store. Third-party developers submit their μ-services to this store, making them available for users. Submitted μ-services can be downloaded from the S2Store and installed inside a DS2OS smart space.

2.4.4 Micro-service (μ-service)

Microservices running inside a smart space make up the third component. Inside a DS2OS, every running software service, besides the VSL, is a μ-service, developed relying on the DS2OS framework. μ-services in this context follow the same principle as apps and can exchange data over the VSL.
Microservices within the DS2OS are small self-contained services, which are meant to be used together with other services in an IoT context. Being the minimal functional unit, μ-services are very powerful and at the same time very flexible. The entire service management system can be built entirely using dedicated μ-services.

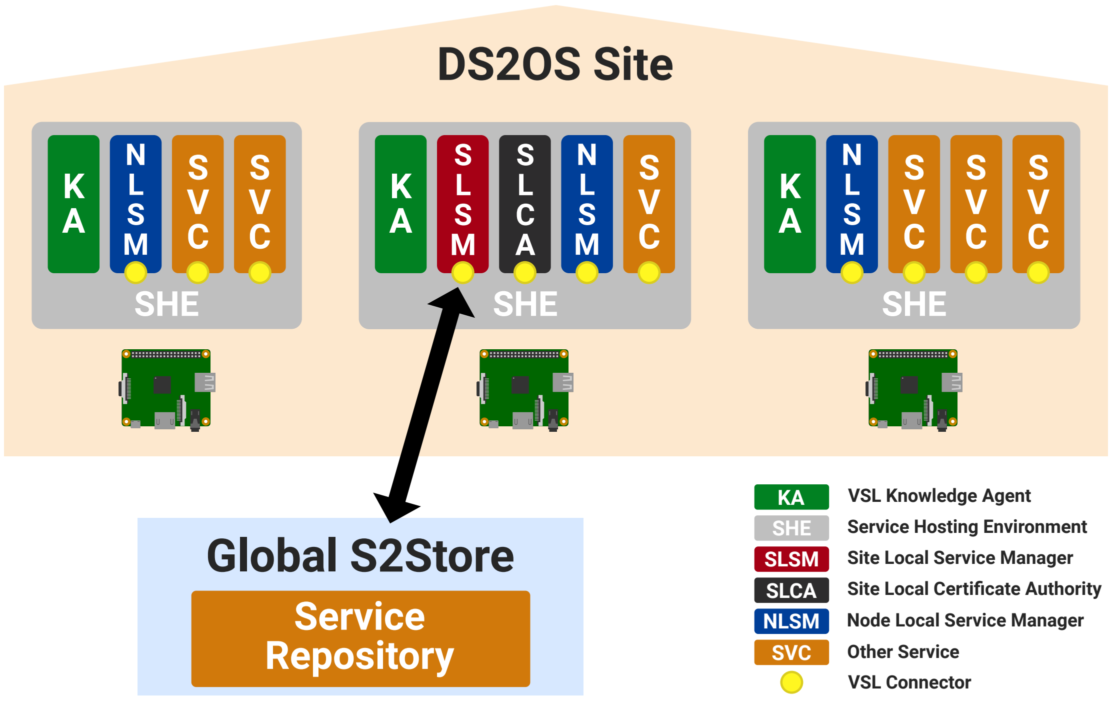


Figure 2.7: Distributed Smart Space Orchestration System (DS2OS) architecture

From an architectural point of view, the service management system considers the components shown in 2.7. Each of these is a specific-purpose μ-service, that enables the management of the smart space by working together with the other services. A detailed analysis of the components follows.

2.4.5 Service Hosting Environment (SHE)

The SHE is a runtime environment for the μ-services, and is unique per every node. Its main function is to interact with the underlying hardware and provide functionalities typical of a HAL. It takes care of (re)starting and stopping the μ-services and interfaces them with the system. μ-services need common interfaces, as they may require access to:

- the VSL knowledge, also used as an interface to other μ-services
- system peripherals
- other hardware resources, such as sensors, bus systems, etc.
- the file system

Dependencies and runtime parameters are provided by the SHE to the μ-services, which run agnostically of the environment they're deployed in. It is the duty of the SHE to start the services in the right order, monitor them and act accordingly, in case of service failure or unexpected behavior.

The SHE may be implemented on top of an OS or provide the low-level functionalities that an OS does. The in-depth study of this component is out of the scope of this thesis, but is extensively analyzed in [58].

2.4.6 Knowledge Agent (KA)

Knowledge Agents are instances run locally on each node that support the VSL middleware, as described in 2.4.2. Leveraging the underlying knowledge sharing mechanisms they keep a partial (or full) local copy of the knowledge of the network. Whenever a specific information inside the knowledge tree is required, it is queried from the node which possesses the requested information.
The knowledge sharing mechanism is fundamental within the DS2OS, as it allows nodes to retrieve information from other nodes. The visibility and access rights of this knowledge is discussed in 2.4.10.

In case of connection loss or intermittent connectivity, the KA will keep running normally. The local knowledge tree will keep getting updated, based on data set by the overlying services. The KA will automatically attempt to reconnect to the rest of the network and then update the shared knowledge tree.

Each node runs only one KA, started by the SHE and managed accordingly in case of failures. Every μ-service started by the same SHE instances connects to the same KA (if needed) and uses it to access context model data. Regardless of the nature of the service, the SHE passively monitors it and collects data needed for management and load-balancing purposes.

2.4.7 Site Local Service Manager (SLSM)

The DS2OS runs on distributed nodes, but instead of having a completely distributed service management, it considers a central trusted entity within the system. This entity is the SLSM, which acts as the orchestrator of the entire smart space. Every other node knows the central entity and relies on it to take management and load-balancing decisions within the smart space. Key responsibilities of the SLSM include:

- keep track of the existing nodes in the network
- download μ-services from the S2Store
- distribute μ-services on target node(s), according to specific policies
- instruct nodes to remove specific μ-services if needed
- manage failure/churn of nodes running critical services
- perform load-balancing and service migration according to optimization strategies

In the basic design of the DS2OS, the SLSM is a unique entity within the smart space and is typically run on the most powerful node, as it requires more computational resources.

2.4.8 Site Local Certificate Authority (SLCA)

The DS2OS enforces service authentication with the VSL, by presenting a valid service certificate. To issue these certificates, a Certificate Authority (CA) is needed. Instead of relying on a global CA, the DS2OS proposes the leverage a dedicated local CA. The SLCA embodies this entity and is unique for each DS2OS site. A more in-depth analysis about certificates is presented in 2.7.

2.4.9 Node Local Service Manager (NLSM)

The last fundamental brick needed for smart space management is the NLSM. This component is more strictly coupled with the SHE and manages the lifecycle of services running on a node.
An NLSM is unique for every node and is always the first μ-service to be started by the SHE.
The main tasks of the NLSM include:

- (un)installing μ-service if instructed so by the SLSM
- automatically start services installed on the node
- automatically stop services before updating them, or if instructed so
- request new certificates when needed (see 2.7)
- monitor the resource usage and behavior of the μ-services running locally
- periodically inform the SLSM about the resource usage
- relay important information (such as critical errors) to the SLSM

The NLSM contains the logic for performing the mentioned tasks, but may not execute all the operations on its own. For example the NLSM takes the active decision of instructing the SHE to start a service. The service processes, however, is not started directly by the NLSM but by the SHE. Although the NLSM and the SHE turn out to be coupled, the logic is still kept completely separate from the system-level operations. More details on this in chapter 4.

It is worth mentioning that on the master node of a site, hosting the SLSM, an NLSM instance is still needed to manage all the running μ-services.

2.4.10 Current Security Concept

The VSL supports communication encryption out-of-the-box. This is achieved by establishing TLS connections between different KAs, as well as between a μ-service and the KA running on the same node.

TLS connections provide confidentiality, integrity and authentication on the communication layer. As a consequence, all data shared between KAs is automatically protected and encrypted.

Digital certificates are used for establishing a TLS channel. More specifically, each agent requires a certificate from a CA. When different agents try to exchange data, they

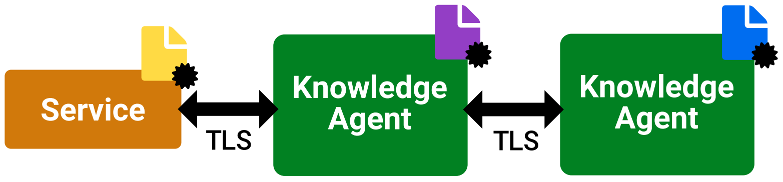


FIGURE 2.8: Authentication between a μ-service and a KA (left) and between two KAs (right) to create TLS connections. Each entity uses a different certificate signed by the same CA.

authenticate one another through their certificates and then establish a TLS connection. For this authentication to succeed, both certificates need to be issued by the same CA.

The VSL also requires services to present a valid certificate when registering a context model. This has two purposes:

- authenticate the service, making sure the certificate was issued by a valid CA
- apply access control rules, expressed as certificate metadata

The VSL is completely agnostic of the management system introduced by the DS2OS. However, it expects all the certificates to be issued by the same CA. Figure 2.8 shows that, even though the certificates on the two KAs and the μ-service are different, the signature has the same origin.

The VSL implements *security by design*, since it protects its most relevant feature as a middleware: communication and knowledge sharing. It is preferable, to build additional security layers on top of a system already providing End-To-End (E2E) encryption. For this reason, this thesis builds upon the DS2OS and aims to provide additional security mechanisms for securing the service management.
The DS2OS security concept, introduced in [59], considers leveraging digital certificates for protecting the μ-services, as well as the entire service management workflow.

The main research questions that this thesis wishes to address are:

<Q.0> *How can a distributed smart space orchestration system, such as the DS2OS, be protected against malicious attacks?*

<Q.1> *How is the introduced security concept backed in the state of the art?*

<Q.2> *How can the security concept be implemented and improved?*

2.5 Internet of Things (IoT) Security

Smart devices are intrinsically computers and are at least as vulnerable to cyber-attacks and faults as traditional computers. Like for any other computing device, security and privacy are a major concern. A thorough security assessment of all existing IoT vulnerabilities is almost impossible to make, but some main attacks and threats identified in literature [53], [5], [19] are worth mentioning.

2.5.1 Attacks on the IoT

For categorizing attacks, the industry and research community refers to the IoT reference model, shown in Figure 2.5. Referring to the seven-layer model facilitates the analysis of threats on a level-basis.
This thesis mainly focuses on applications running on edge nodes within smart spaces. While communication between edge nodes and remote servers is not out of the question, the analyzed scenario considers users to directly manage their smart devices and services. This leads to a simplified IoT reference model, containing only 3 layers: a *physical layer*, a *connectivity layer* and the *application layer*, which may include all the upper layers collapsed into one (i.e. data storage, aggregation and even collaboration between different applications).

Name	**Short**	**Description**
Confidentiality	C	Concealment of information
Data Integrity	I	No improper or unauthorized change of data
Availability	A	Services should be available and function correctly
Authenticity	AU	An entity is who she claims to be
Accountability	AC	Identify the entity responsible for a communication event
Controlled Access	CA	Only authorized entities can access certain services or information

Table 2.1: Reference technical definition of security goals

The security goals considered in table 2.1 are used as a reference for the following analysis. Each attack on the IoT may impact one or more of these defined security goals.

Table Table 2.2 shows a summary of the major threats and countermeasures that can be found in the IoT.

Attack	**Layer**	**Impact**	**Countermeasure**
Hardware Trojan	Phy	All	Controlled circuit design and production process
Node tampering	Phy	I,A	Restrictions to physical device access, secure boot
Node impersonation	Phy	AU	Authentication mechanisms
Eavesdropping	Net	C	Communication encryption
Man in The Middle (MiTM)	Net	All	Secure channels, with authentication, encryption and integrity protection
Routing	Net	C,I,A,AC	Encryption, advanced routing techniques
Sybil	Net	All	Enforced authentication, encryption, advanced routing techniques
Unauthorized Conversation	Net	AU,CA	Authentication mechanisms, access control rules
Cryptanalysis	App	C	Strong encryption schemes
Virus, Malware and Spyware	App	C,I,A,CA	Authentication, controlled access, application integrity protection, anti-malware
Malicious code injection	App	All	Authentication, controlled access, application integrity protection
Protocol	App	C,A,AU	Auditing and testing
Denial of Service (DoS)	Any	A,AC	Authentication, firewalls, controlled application management
Social Engineering	Any	C	Protect confidential information, promote awareness
Side-channel	Any	C	Restricted physical access, metadata encryption, de-patterning transmission

Table 2.2: Attacks on the IoT physical layer and countermeasures

The typical countermeasures adopted in research works and industry solutions for the IoT include (see chapter 3):

- protection of the IoT devices, relying on integrity checks and the encryption of the device itself;
- enforcing the usage of secure channels for securing the communication between IoT devices;
- use device certification, by pinning certificates to the software running on them;
- to address application security, existing app ecosystems leverage code-signing and automated analysis techniques for protection (analyzed in subsection 2.6.3).

2.5.2 Threat assessment

While each one of the analyzed layers should be protected properly, the presented range of attacks is too wide to be addressed in a single solution. The basic assumption is that the attacker may be the network, or someone with the possibility to inject malicious applications or tamper with existing applications.
In light of the identified threats, this assessment considers feasible approaches, constraints and defines the target security goals.

Physical Layer

The main constraint on the physical layer is given by the low resources of the involved devices. Many IoT devices do not support powerful cryptographic primitives, such as Advanced Encryption Standard (AES) or Secure Hash Algorithm (SHA) algorithms.

Physical attacks are not considered within the scope of the thesis, since they are unrelated to the top-level service management. However, some basic assumptions for the analysis will be:

- devices have at least basic cryptographic capabilities
- attackers cannot physically access the vulnerable devices

Network Layer

The ideal protection on the network layer is the usage of secure channels for communication. By authenticating and encrypting messages (e.g. using TLS), data integrity, authentication and confidentiality are guaranteed. The scenario targeted in this thesis relies on the existing VSL, which provides this functionality already (see subsection 2.4.10). Therefore, many of the attacks targeting the network layer are already protected against.

Application Layer

On the application layer, the chosen IoT scenario based on apps, any kind of software-based attack is to be considered.
Software applications are the main source of security issues, as they offer a wider range of functionalities to exploit. Vulnerabilities may stem from system software, edge computing algorithms, high-level applications or any other kind of computerised software.
It is impossible to determine defenses for all attacks on applications, as they depend on

the nature and behavior of the specific application. The main protection mechanisms against malicious code execution are briefly described.

Data security can be enforced with *authentication, encryption and integrity* mechanisms. This applies to sensitive data and application binaries. Effective encryption and checksum mechanisms ensure confidentiality and integrity of the device's data, nullifying data theft and manipulation implications.
Proper *access control lists* also enforce authorized access to data, ensuring data privacy.
Firewalls can protect the system from undesired external entities, and help preventing DoS attacks.
Anti-malware software prevents malicious code to be installed and/or launched. It is, however, not a suitable solution for the IoT. This is due to the heterogeneous nature of the devices, the additional cost, the overhead and the required update frequency of such a solution.

The relevant problems identified in this section are:

<P.5> IoT nodes are vulnerable to unauthorized access to resources, leading to data integrity violations and data theft

<P.6> Software for the IoT is not authenticated nor verified, and cannot be trusted

<P.7> Malware and code injection lead to violations of data integrity, confidentiality, availability and resource access

2.5.3 Security Goals

IoT devices running services inside a smart space face all the vulnerabilities mentioned in 2.1 and 2.5. Based on the threat assessment in subsection 2.5.2, the scope of the security that this thesis aims to provide becomes clearer. The main addressed security goals, which are of the utmost importance in the analyzed scenario, are:

- application **integrity** [23]
- **authenticity** of applications and nodes
- **controlled access** [22] of IoT resources and interfaces

The countermeasures analyzed in 2.5.2 are mainly based on cryptographic algorithms, digital signatures, access control policies and automated analysis algorithms.
Given this consideration and the target security goals, several state of the art approaches need to be analyzed more in depth.

2.6 App Security

The architecture of the DS2OS, analyzed in subsection 2.4.1, resembles in many ways a typical app store paradigm, which is widely used in the industry today. This paradigm was born in the first decade of the 21st century, when smartphones began being mainstream.

The term *App Ecosystem* is used to refer to all activities and infrastructures that surround applications targeting a specific device category. This includes the development of the apps, the server infrastructure, distribution platform, analytics tools, advertising, monetisation systems, user feedback, support and more.

Tapping into the knowledge of modern app stores and existing app management systems helps identifying the required security mechanisms for the DS2OS. This section analyzes security features provided by existing app ecosystems. The analyzed features lead to the system requirements or may serve as foundation for fine-tailored mechanisms to be applied to the DS2OS.
An in-depth comparison with existing app ecosystems will follow in section 3.2.

2.6.1 App Stores

An *App Store* is a digital distribution platform, meant to store apps and offer them to its customers from a centralized entity. The set of supported applications typically targets a specific platform or type of device.

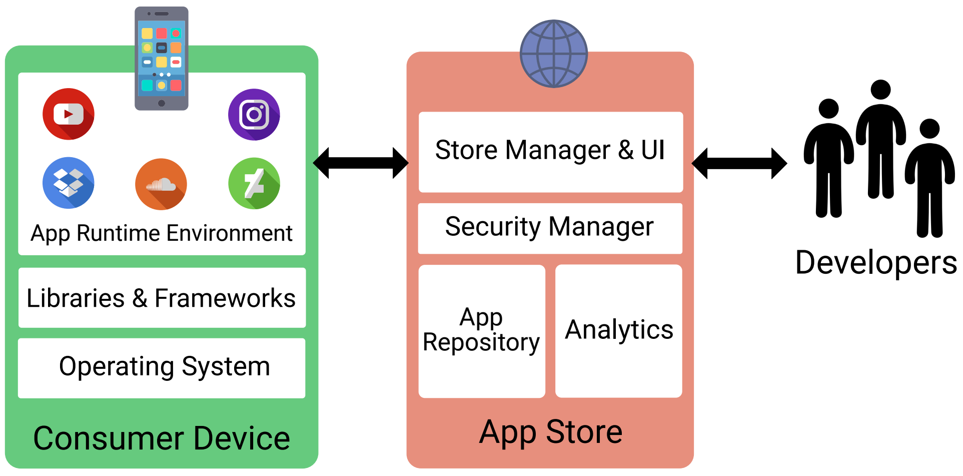


Figure 2.9: Architecture of a typical App store

An example architecture of a typical app store is depicted in Figure 2.9, with no reference to a specific vendor.
Applications developed by third party developers are uploaded on a global platform. The platform is in charge of storing, distributing and keeping apps secure from external influences. The platform also supports additional features, such as managing analytics and data generated by the consumers, which is not relevant for this thesis. Consumers may download app on their devices via a UI, offered by the store.

The introduction of digital distribution platforms solves several problems:

- Users don't have to cross-search the Internet for the desired application anymore, but can find it on a centralized platform
- Privacy and security are enforced by the store. No malicious apps can be downloaded and the user's data is not leaked to other parties.
- Applications offered by a store are considered to be trusted, as the company behind it earns money for it. By extension, the store itself is a trusted party

The architecture of the DS2OS heavily relies on the S2Store component. The S2Store contains the service repository where all available μ-services are stored and can be downloaded from.

The next subsections will focus on several security-related aspects introduced by app stores. In particular, how the integrity of an app executable is guaranteed, how the authentication mechanism works, how the access rights are defined and enforced, and finally how dependencies are handled differently in each system.

2.6.2 App Distribution

The distribution process of an App is essentially the same for most existing app ecosystem (see 3.2).
Each store is part of a unique App ecosystem and therefore offers a different Software Development Kit (SDK). Third-party developers have to stick to the supported programming language(s), file formats and guidelines imposed by the ecosystem when developing an application. Apart from platform-specific practices, UI design and limitations, the concept behind each ecosystem is fundamentally the same.

Developers upload application packages to an app store using a dedicated UI. Each package contains the app and additional metadata, required by the store. The metadata is typically generated via the SDK or other tools provided to developers. The store makes sure that only authorised developers can do so (see 2.6.3).

Apps submitted to the store undergo a rigorous review and verification process, to make sure that no consumer may be harmed by them.
Once the application package has been accepted to the store, it is kept inside a repository, accessible to consumers.
The consumers install an application from the store to a target device. To do this, they access the store UI from the very device on which they wish to install the application. Some applications have restrictions and dependencies, such as a specific target device (e.g. a tablet rather than a smartphone), other hardware or software constraints. Dependency management is addressed in 2.6.5.

The store seamlessly takes care of performing the necessary checks and makes sure that the target device meets all requirements and dependencies. If successful, it allows the requester to download the application bundle.

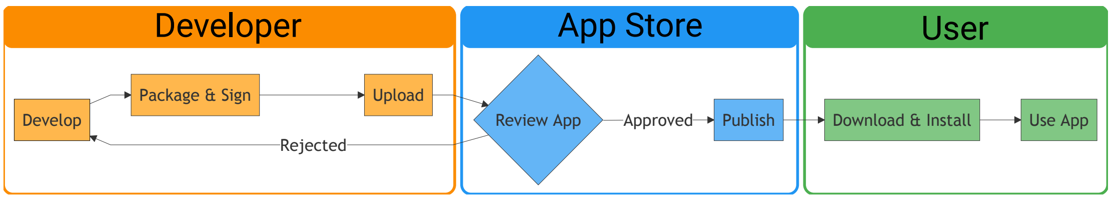


Figure 2.10: Exemplary app distribution process

Once an app package has been downloaded, it is unpacked in local storage according to the installation mechanism. At this point integrity check mechanisms are in place, needed to guarantee, that the package was not tampered with in any way (see subsection 2.6.3). After installing an application, the consumer is able to use it and doesn't have to communicate with the store any longer. The simplified workflow is shown in Figure 2.10.

The purpose of the store, is to function as a centralized secure repository and allow scalable distribution of applications, safeguarding consumers from malicious software. A store acts as a trust anchor, and users assume that the software coming from it does not contain malicious code. In a way, stores must ensure accountability for the distributed applications. Stores enforces security policies and perform code reviews, verifying the quality and benign purpose of applications.

Distributed Target Devices

The fundamental difference between the DS2OS, and traditional app ecosystems (see 3.2) is that the former does not deploy apps on the same device they were downloaded on. In a typical app ecosystem (e. g. iOS), the device downloading the app is also the device on which the app itself will run.

Within a DS2OS site, nodes do not download a μ-service from the store directly, with the exception of the SLSM. The SLSM is the only entity responsible for orchestrating the smart space. Its responsibilities include dynamically distributing μ-services to other nodes within the site. As a consequence, it is the entity in charge of downloading every μ-service, that will be run within the DS2OS site.
Figure 2.11 shows the two steps needed to deploy a μ-service, when an administrator (user) instructs the orchestrator to install a specific service:

1. the SLSM downloads the μ-service, chosen by the user, from the S2Store and stores it locally;
2. according to internal policies and load balancing strategies, the downloaded μ-service is sent to a suitable node in the site. The node manager (the NLSM) running on that node installs the μ-service and runs it.

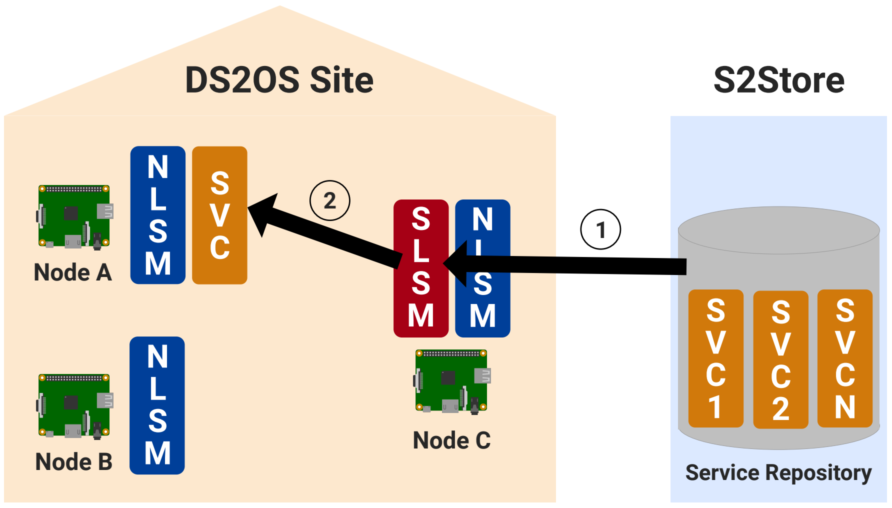


Figure 2.11: Microservice installation procedure within a DS2OS site

The described procedure is not initiated by the NLSM itself, but is triggered by the SLSM. Still, the SLSM acts like a trusted store entity towards the NLSM.
Regular nodes act as slaves within a DS2OS site and do not contain decisional or management logic. The only entity taking decisions within the site is the SLSM, which is indirectly controlled by the user.

In terms of usability, in a distributed scenario, such as the DS2OS, it would be inadequate to have every node install a μ-service directly from the store. There are two main reasons for this:

- centralised management of the smart space is not possible, if every node communicates with the store independently;
- every deployment would require manual intervention, which goes against the idea of a modern smart space, where nodes are unattended most of the time.

Compared to traditional app distribution, the proposed mechanism is novel and requires additional management to be effective. Such management logic is not within the scope of the thesis and is already addressed in [12] and [58]. Rather, the thesis aims to neutralize potential new attack vectors introduced with this novel distributed service management.

Some potential risks and factors that need to be kept in mind in a distributed environment are:

<P.8> Attackers may impersonate the SLSM and trick other nodes into accepting a forged service bundle, containing malicious code. Nodes must be able to identify and trust the SLSM

<P.9> Attackers may impersonate an NLSM and obtain a service bundle from the SLSM, gaining unauthorized access to resources

On one hand the SLSM needs to take on the role of the store, guaranteeing that an installed service bundle is identical to the original one downloaded from the S2Store and in no way modified.
On the other hand, the SLSM must be wary of other nodes within the local site. When deploying a μ-service to a node, the SLSM needs to be sure, that the target node belongs to the smart space and is not an impostor.

Several other things need to be considered in a self-managing smart space. When transferring a service package to a different node, it is necessary to verify the requirements, dependencies and access rights of the service. Especially when nodes are heterogeneous IoT devices, hardware capabilities and resources are subject to change. The administrator of a DS2OS site may also have set up different constraints and permissions on different nodes. This requires proper management, since nodes must not run services, if they are not authorised to do so.

To sum up the problems, the introduction of a completely distributed network makes the deployment of applications on heterogeneous nodes intrinsically more difficult. Not only is it necessary to manage the installation, uninstallation and service migration process (see section 4.3), but certain security mechanisms need to be put in place to overcome the introduced risks.

2.6.3 Code Signing

Applications are usually distributed as bundles (see 3.2), containing an executable, along with other metadata and resources. *Code signing* is a well-known technique that provides both data integrity and authentication of the source. The *integrity protection* proves that the code was not modified, while *authentication* allows to identify who signed the code.

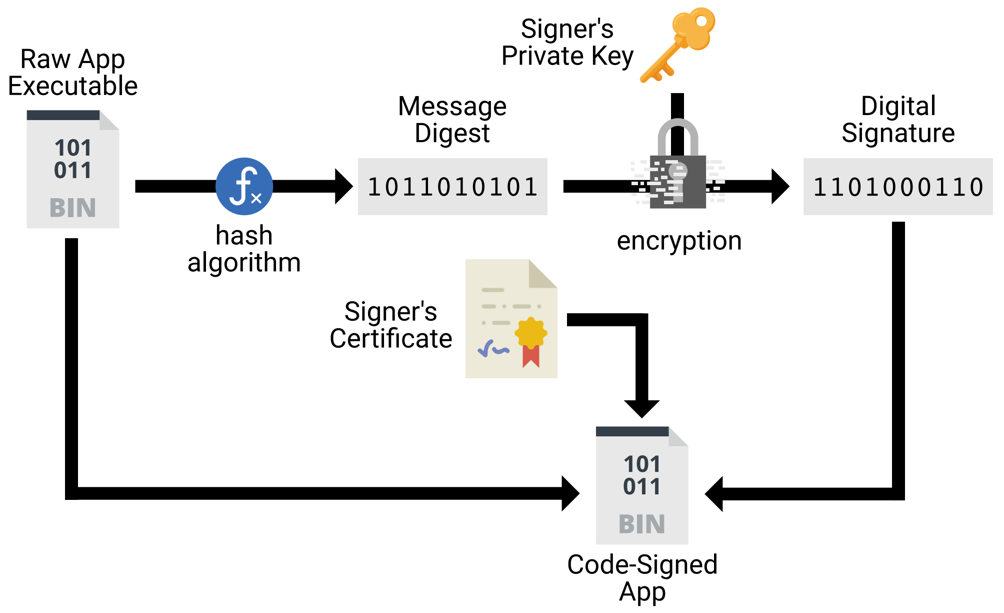


Figure 2.12: Generic code signing mechanism

A code signing procedure is visible in Figure 2.12. Verifying a signature involves hashing the original application and checking whether the obtained message digest is equal to the one contained in the code-signed application. The verification is visible in Figure 2.13. Message digests are protected thanks to encryption. Public Key Infrastructure (PKI) key-pairs are used to ensure authenticity.

Code signing is by far the most widely used and effective technique to protect an application bundle. As long as the target device supports basic cryptography and digital certificates, code signing μ-services is a very valid solution.

Integrity Protection

Skilled attackers are capable of running arbitrary code by changing a very limited amount of bytes inside an executable. This is one of the major risks that app stores have to mitigate.

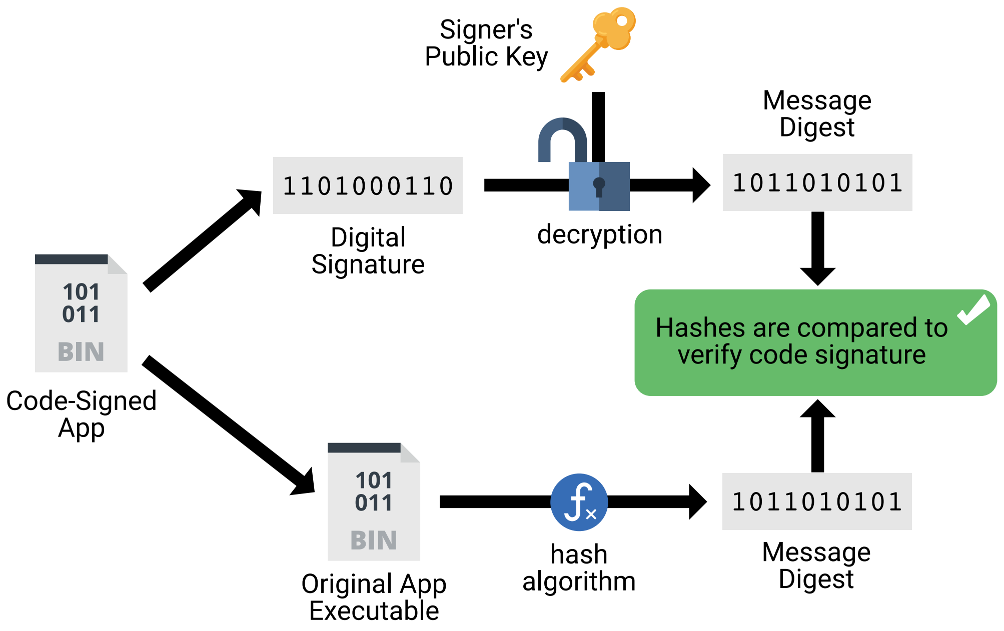


Figure 2.13: Code signing verification mechanism

Compromised applications can cause damages to devices and customers alike, leaking private data to third parties or arbitrarily modifying/deleting existing data. This potentially leads to the violation of every security goal.
Any modern app store is considered a trust anchor, and should not distribute applications containing dangerous code. Especially in an ecosystem, where apps are paid for, it is undesirable to deliver any malicious application to the customer. The non-expert customer assumes that any application coming from the app store, as well as the installation process, are secure. To make a comparison, untrusted sources (e. g. web pages) do not give any guarantees, and malicious code is a constant risk.
The reasoning made for application executables also applies to any sort of additional metadata attached to it.

IoT nodes running third-party services are affected by the mentioned problems, like any other application. It should not be possible for any attacker, to modify a service package from the moment the developer uploads it to the store until the moment it is executed on the target device. It is also worth mentioning, that an application cannot be considered as not compromised, if already installed on the target device. Necessary precautions must be taken, to avoid the same application (binary or metadata) to be modified while on the device.

Identified problems and integrity risks that need to be mapped as requirements are:

<P.10> Application executables are prone to binary modification from attackers, leading to arbitrary code execution

<P.11> Unprotected service metadata may be subject to tampering

<P.12> Attackers may modify the service package at any point during the distribution process

<P.13> An application installed on a device may be modified, leading to to arbitrary code execution

Code signing relies on the most obvious mechanism for securing the integrity of both executable and the attached metadata: the checksum (or hash).
Every major app distribution system today relies on code signing techniques to ensure that:

- the application package does not contain error and was not tampered with;
- the application packages can be distributed along with a signature, which belongs to a CA or to the developer of the application.

Authentication

<P.14> Third-party applications may originate from untrusted sources and contain malicious code.

Code signing has the purpose of providing code authenticity, therefore guaranteeing the source of an application.
Code signing relies on developers signing their apps using their *private key.* This implies the support of traditional Public Key Infrastructure (PKI).

App stores, as well as target devices and consumers can verify the source of the application by checking the signature (shown in Figure 2.13). By assumption, before submitting app to an app store, a developer has already uploaded their *public key* to the store.

2.6.4 Runtime Environment (RTE)

Besides providing integrity checks and authorization policies, applications should also be run in a safe environment. Proper RTE control can protect against privilege escalation and undesired damages to unprotected parts of the system.
Modern app ecosystems enforce app *sandboxing* techniques, to limit the access the app.

The focus of the thesis is not to provide a safe RTE, as it is the goal of the SHE. Refer to chapter 4 for more information.

2.6.5 Dependencies and Access Rights

Applications often have dependencies. These may be hardware or software dependencies. For example, an application for taking photographs requires access to a camera. The same way, an application that analyzes the room temperature needs access to the room temperature sensor, or a service that provides this measurement.
Dependencies between applications are managed in different ways, depending on the target platform. In some cases the control is completely transferred to another application. In other cases the application leverages an interface, offered by another application, to get access to more functionalities.
The ecosystems analyzed in section 3.2 propose different approaches to this problem.

In comparison, the DS2OS solves dependencies with a custom mechanism, based on *Context Models* (introduced in 2.4.2).
Each μ-service relies on a context model [59] to represent its own data and make it available to other services over the VSL.
Developers declare the context models for their services using XML notation. A context model contains a tree hierarchy which represents the resources/properties of the service.

```xml
<temperatureService type="basic/composed">
        <currentTemperature type="basic/composed" reader="home"
         ↪  writer="">
                <celsius type="basic/number" />
                <fahrenheit type="basic/number" />
        </currentTemperature>
        <sensor type="basic/composed" reader="admin" writer="">
                <model type="basic/text" />
                <firmwareVersion type="basic/text" />
        </sensor>
</temperatureService>
```

Listing 2: DS2OS Context Model for a temperature service, with defined access rights

The example in listing 2 is an extension of the context model presented in 2.4.2. The context model contains a temperature value in Celsius and Fahrenheit, as well as some information about the temperature sensor. Services in charge of adjusting the room temperature, via air conditioners or heaters, require the information provided by the temperature service. This is considered a dependency. Service bundles requiring data provided by additional services must declare this within the manifest.

An important functionality of the DS2OS is to deploy μ-services according to their dependencies. If a dependency cannot be resolved, a μ-service should not be run. Hence, if the temperature service was not available within a site, the temperature management service could not be deployed there either, because of the missing dependency.

The context model shown above describes a temperature service, with additional access rights set.
Access rights define what resources and dependencies an application is authorized to access.

<P.15> Directly accessing hardware resources or other apps is a renowned cause for privacy and security breaches

<P.16> Unauthorized modification of access rights may give attackers read/write access to arbitrary resources inside the smart space

<P.17> Individual access rights for every service are not intuitive. Generic access groups should be presented to the user instead

Every major system, capable of running software, enforces *access control*. Major app ecosystems do the same, in the form of *user permissions*. Only users are allowed to modify these permissions. Once set, user permissions are stored in a read-only directory, under the control of the system. This makes sure that the access rights cannot be modified by attackers at any given time.

Services within the DS2OS leverage the *Context Model*-based mechanism described in subsection 2.6.5, which allows to manage access rights for each specific dependency (introduced in subsection 2.4.2).
The VSL guarantees that, given the correct access rights, a service may access the data defined in the context model of another service.

Access rights are subject to change at any given time. These changes are typically user-initiated and the user expects them to be applied effective immediately. This is especially challenging in a distributed scenario and leads to the following necessities:

<P.18> Access rights revocation or changes need to be propagated to all affected nodes

<P.19> It must be possible to update rights changes on nodes that are out of reach (due to intermittent connectivity) as soon as possible

2.7 Digital Certificates

In light of the security concept presented in 2.4.10, an analysis of digital certificates is necessary.
In computer science, digital certificates allow to uniquely identify an entity, be it a person, an organization or else. This is fundamental in order to establish secure communication channels with other services over the internet, or perform general-purpose authentication.
The concept was borrowed from real-life scenarios.
An real-life example are identification documents, e.g. passports. These may only be issued by globally recognised authorities (governments in this case), which are considered trusted by third parties. If a document is signed by a trusted entity, then the document is, by transitive property, also trusted.

A digital certificate is issued by a trusted Certificate Authority (CA). To prevent counterfeits, strong cryptographic primitives are used. These make sure, that it is computationally infeasible to steal someone's identity.

Digital certificates are also called *Public Key certificates*, as they guarantee that a certain entity is tied to a unique public key. Private and public keys are at the core of modern PKI.

<P.20> Digital certificates require cryptographic operations and are have a considerable size. Their usage should, therefore, be optimized.

2.7.1 Application Certificates

The practice of pinning a digital certificate to a device or an application is well-known, both in research and in the industry ([71], [42]). In the proposed scenario (see section 2.4), each μ-service already needs a valid certificate to connect to the VSL. Therefore, a valid approach would be to leverage the same functionality to also satisfy other security goals.

2.7.2 Signed Metadata

Digital certificates support additional information to be attached to it, as signed metadata.
The advantage of using a single certificate and attaching metadata to it is manyfold:

- authentication with the VSL is guaranteed

- code-signing of a μ-service can be combined with the service certificate
- access rights can be embedded as metadata

Since each certificate is issued for a specific service, the control over the metadata would be much more granular. Ultimately, application integrity, authenticity and access control can be achieved.

<P.21> Different service management entities and services may require different metadata.

The defined metadata is not required on every node and at every stage of the service management. For example, user-defined access rights are not available by a user when the developer submits a service. Therefore, the required metadata must be considered carefully.

2.7.3 Certificate Authority

A trust anchor is needed, as soon as authentication mechanisms come into play. For digital certificates, such an entity is called Certificate Authority (CA).
CAs play a critical role in today's internet. They are in charge of issuing new digital certificates for servers and service providers. Clients trust the certificates emitted by a CA, because the CA acts as a guarantor for the service provider.

For the proposed scenario, smart spaces need to rely on a CA for issuing valid application certificates. Two options are possible:

1. leverage a global CA
2. employ a local CA for each smart space, to improve autonomy

The first option is clearly less scalable, even considering an intermediate authority. The workload of issuing certificates for every existing smart space can become a huge bottleneck. Furthermore, a certificate should be issued specifically for a user/device and not have global validity. A certificate issued in a smart space should not be accepted as valid in any other Smart Space. Otherwise, this could lead to certificate spoofing in any network.

The second option seems more fit for this scenario, but imposes the overhead of managing a local CA.

Summing up the identified problems:

<P.22> Issuing service certificates in the DS2OS requires a dedicated trusted entity to take care of it

<P.23> Certificates issued by a CA must be tied to a smart space and managed appropriately, to avoid spoofing

2.7.4 Certificate Revocation

Solving problems [P.18] and [P.19] inside a distributed system requires a dedicated mechanism.

When metadata is changed, it is necessary to propagate the changes throughout the smart space and ensure consistency on every node.
In a certificate-based scenario, updating metadata can be tackled in two steps:

1. existing metadata needs to be invalidated
2. new metadata needs to be applied automatically

Invalidation is achieved through certificate revocation. In literature, there are several documented approaches for successfully revoking a certificate:

- Certificate Revocation List (CRL) [20]
- Online Certificate Status Protocol (OCSP) [67]
- OCSP Stapling [30]
- Certificate Revocation Guard (CRG) [33]
- Certificate expiration [66]

All these approaches target Secure Sockets Layer (SSL) certificates, commonly used for web browsing.

Certificate Revocation List (CRL)

CRLs were originally used by CAs to publish revoked certificates. Clients would download them and compare them with current server certificates.

While this approach scales poorly in the internet, it would be a valid approach for a smart space. A smart space is expected to host a reasonable amount of devices, connected within the same network. Therefore, the CRL scalability should not prove to be an issue. Valid counterarguments to this approach are:

- It is hard to predict how the size of a CRL would increase over time
- the complexity required for managing CRLs can become high

- to make a revocation effective, nodes have to check for CRLs updates often

Custom solutions built-in in browsers, such as CRL Sets by *Google*, use an intermediate server instead of a CA. While this does improve performance in the internet, it does not offer any real advantages over traditional CRLs in a smart space.

Online Certificate Status Protocol (OCSP)

OCSP is an improvement to CRLs that checks if a certificate has been revoked, thanks to an additional request to the CA during a TLS handshake. While this approach is more performant, an issue still arises: the CA is queried only during a TLS handshake by default. To make a revocation effective, a service has to re-establish a connection, in order to get this information. Periodic checks to overcome this limitation can lead to the same issues discussed for CRLs.

OCSP Stapling

OCSP Stapling is an optimization for the web infrastructure, in which the revocation information resides also on the server, and not just the CA. With this approach, the smart space orchestrator would have to fetch and maintain the list of revoked certificates. This is a valid option, although it requires to implement additional protocol logic for OCSP.

Certificate Revocation Guard (CRG)

CRG is a solution introduced in [33]. The research approach involves an entity, called CRG, intercepting a certificate during a TLS handshake between a client and a server. The entity checks whether the certificate has been revoked, either from a local database or by fetching it directly from the CA (either CRL or OCSP can be used for this purpose). If the current certificate has been revoked, the client is notified of this.

This option outperforms solutions based on CRL and OCSP, both in terms of bandwidth and used storage, but is not as widely used. This approach is valid, and the main counterarguments are:

- increased software architecture complexity
- the new CRG entity needs to be trusted as well, and requires further protection

Certificate expiration

The last approach involves leveraging a built-in property of digital certificates: the expiration date. Every certificate has a validity period, defined by the issuing CA. Once the certificate has expired, it is no longer valid and a new one needs to be issued.

This built-in feature is very powerful, as it ensures an automatic invalidation of a certificate, even if a node is not reachable. DS2OS services rely on certificates to be run.

<P.24> If a certificate is invalid, they cannot be run, nor can they connect to the VSL.

A controlled certificate lifetime can also improve security and reduce the damage caused by key compromise.

The obvious downsides to this approach are:

- Periodically renewing certificates, after they have expired, has a cost, involving computational resources and network bandwidth
- Concurrent certificate renewals can lead to bottlenecks
- A service with an invalid certificate cannot communicate with the VSL

2.8 Authentication Tokens

Token-based authentication is a widespread technique. The most common use-case for authentication tokens is to allow an entity temporary access to a resource. Instead of requesting a resource from a server every time it is needed, the idea is to verify the a client's credentials only once.

The token concept is similar to real-world scenarios, in which a complete authentication can become burdensome if performed continuously. An example is a conference, where participants first register with their own government-issued ID. Upon registration, they receive a conference-specific badge (i. e., a token), with its own unique ID. This process is an authentication, where temporary credentials are generated. These credentials are valid only at that specific conference and for one specific person. Throughout the course of the conference, each participant uses their token to get in and out, without having to show the ID they showed on the first access.

The main digital authentication token protocol nowadays is described in subsection 2.8.1. The most widely used format for tokens is described in subsection 2.8.2

Workflow

The workflow for digital authentication tokens is straightforward:

1. A client sends its credentials to a server
2. The server verifies the received credentials
3. If credentials are valid, the server sends a token back to the client
4. For every subsequent request to the server, the client will embed the token inside the request itself
5. The server can verify whether the received token is valid for every received request

Tokens are simple byte sequences with arbitrary length that have meaning within their own authentication system. Tokens are generated using a cryptographically strong Pseudo-Random Number Generator (PRNG), together with the metadata needed for access control. The data is integrity-protected thanks to a server-generated signature (e. g. [31]), embedded into the token.

Advantages and Disadvantages

The main advantages of authentication tokens are:

- *statelessness*, which makes the entire approach scalable
- support for *distributed μ-services*, as a token generation is decoupled from the token generation. This means that one service may generate a token, which is used for many other services. Authentication against multiple servers is also possible
- *fine-grained access control*, since a different token can be generated for every resource a client requires

The main security flaw of tokens, is that they may be stolen. The same problem also occurs for certificate-based solutions:

<P.25> An unauthorized entity with someone else's token/certificate gets access to a resource, which should be inaccessible to that entity

This problem is dealt with by using tokens with short validity. If the expiration of a token is very early, then the impact of token theft can be limited to some degree. When a token expires, a client needs to request a new one. This process is typically seamless and automated.

2.8.1 OAuth 2.0

OAuth2 is an open standard protocol (framework) used for resource access delegation [15].

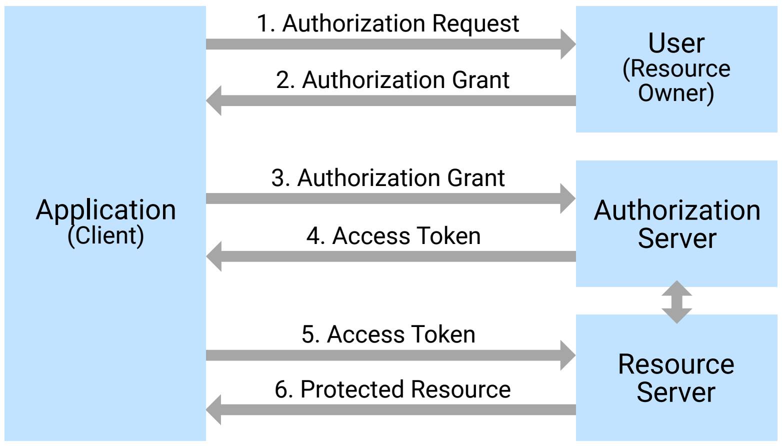


Figure 2.14: Client resource authorization workflow with OAuth2

OAuth2 defines 4 roles: the *resource owner*, the *resource server*, the *authorization server* and the *client*.
The central entity introduced in this solution is the *authorization server*, which takes care of generating tokens and storing them.

Figure 2.14 shows the protocol used for granting access to a resource. The first four steps are required the first time a client needs to access a specific resource.
After obtaining a valid access token, every time the client requires the resource, only steps 5 and 6 are performed. When receiving a request with access token, the *resource server* delegates the verification of the access token to the *authorization server*.

Authorization servers issue two types of tokens:

- *access token*, which are used to grant access to a protected resource
- *refresh token*, kept by the client and used only for requesting a new token, after the previous one expired. This type of token is not always used.

Both tokens have a limited validity in time and are integrity-protected.

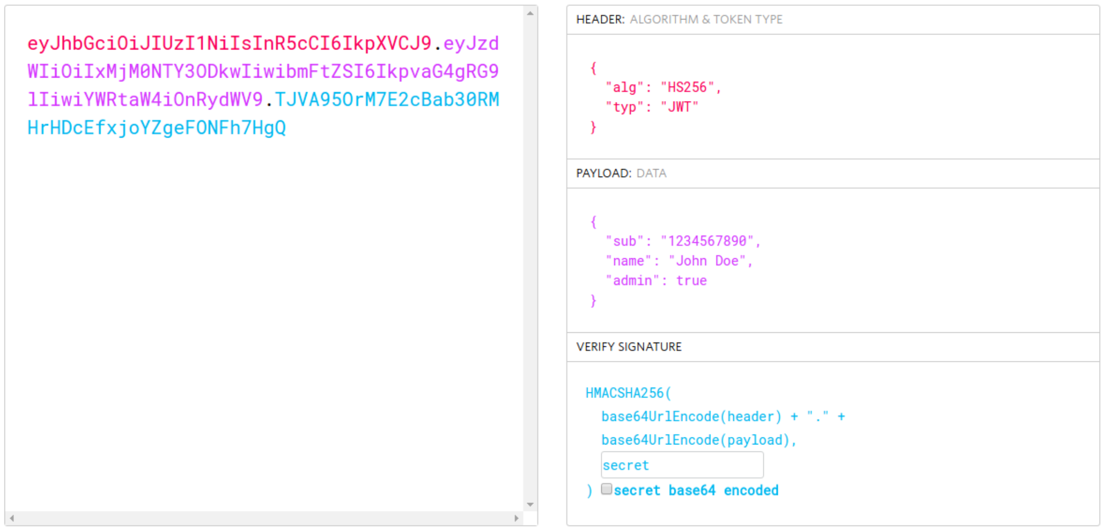
2.8.2 JWT

JSON Web Token (JWT) is an open standard [49] defined for transmitting authorization tokens in a simple JSON format.

JWT have a very simple structure, with three fields separated by a dot:

- The **header** contains information about the used algorithm
- The **payload** includes the claims of the token. Registered claims, such as expiration time and token issuer are also supported [8]
- The **signature** is just a pseudo-random sequence of bytes, generated by signing the encoded header and payload with a secret key. Tokens are signed either using symmetric shared keys (e. g. [31]) or PKI infrastructure.

For HTTP compatibility, tokens are *base64* encoded. An example is visible in Figure 2.15.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzd
WIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RM
HrHDcEfxjoYZgeFONFh7HgQ
```

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) ☐secret base64 encoded
```

FIGURE 2.15: Example of a JWT token. On the left, the encoded token. On the right, the three decoded fields.

JWT may be used together with the OAuth2 protocol described in subsection 2.8.1. Custom applications and internet services consider using dedicated API on their own server to manage JWT tokens (this functionality is identical to the one considered for authorization servers). Web-apps have the advantage of being able to embed a token inside the *Authorization* header of any HTTP request. As long as the token is valid, the client will be granted access.

The benefits of using OAuth2 and JWT together are their *portability*, thanks to the well-known JSON format, their *simplicity* and their *scalability*, as they can be used in the internet.

2.8.3 Authentication for the IoT

While tokens were originally built for the web, to replace sessions IDs, these are now used for any kind of application (usually still over HTTP).

Tokens are good potential candidates for IoT access control as well. For example, solutions such as [10] and [4], consider using tokens for MQTT authentication. Both the JWT and OAuth2 approaches are valid, as they are lightweight, scalable and cross-platform. This closes the gap between the heterogeneity of IoT devices and traditional computers.

Similarly to the certificate-based approach, a token-based solution requires a dedicated authorization server, for distributing and managing access tokens. The other problem, introduced with tokens, is the necessity to embed the authorization code in every request to the resource server.

2.9 Requirements

For this thesis, a custom variant of the revocation based on certificate lifetimes is chosen. The main reason for this choice is the possibility to leverage a built-in functionality, without having to add further complexity on top. The approach is also more documented in literature (see chapter 3).
The expected outcome is a solution with very low complexity, using simple and consistent security mechanisms.

This section translates the problems, encountered in the analysis, into functional and non-functional system requirements. For every defined requirement, the addressed problems are referenced.

2.9.1 Functional

Since the scope of the thesis is to ensure specific security goals (see subsection 2.5.3), the functional requirements are divided in respective sections.

Integrity

Service integrity is the first security goal to be achieved. Addressing the problems identified in 2.6, the following requirements are defined:

<R.1> Service bundles uploaded to the S2Store cannot be modified or tampered with. This includes the executable, the context model and the additional metadata added to the service [P.7] [P.6] [P.10] [P.11] [P.12]

<R.2> Service bundles downloaded from the S2Store to a smart space must be integrity-protected during the entire distribution process. This includes the executable, the context model and the additional metadata attached to the service. [P.7] [P.6][P.10] [P.11] [P.12] [P.13]

<R.3> Services installed on a node, along with their metadata, must be protected against modification and tampering [P.7] [P.10] [P.11] [P.12] [P.13]

<R.4> Service access rights and any site local metadata must be protected from unauthorized modification [P.5] [P.15]

Authentication

The second important security goal is the authenticity of the service and its source. Additionally, the identity of the nodes within a smart space must also be verifiable, to avoid spoofing and misuse of access rights. The problems identified in 2.6 lead to the following requirements:

<R.5> Microservices must be signed by developers in order to be submitted to the S2Store [P.6] [P.14]

<R.6> Only μ-services certified by the S2Store can be downloaded to a smart space [P.6] [P.14]

<R.7> Only services signed by a local CA can run within a local site [P.6] [P.8] [P.22]

<R.8> A local CA can issue certificates only for services downloaded by the user from the S2Store [P.9] [P.6] [P.14] [P.23]

<R.9> Nodes must be authenticated with the smart space orchestrator to obtain new certificates [P.9] [P.25]

<R.10> Service certificates shall be periodically renewed, to force identity verification of nodes [P.25]

Access Control

The VSL automatically checks the validity and applies access rights when services attempt to access a context node. To enforce access control and allow users to modify rules at will, the following requirements are defined:

<R.11> Developers can define access groups within a context model, for improved access rights management [P.17]

<R.12> Access rights for a specific μ-service are defined interactively by the user within a site [P.15] [P.21]

<R.13> Access rights are bound to a specific service and can only be modified by the user [P.15] [P.5] [P.16]

<R.14> When access rights for a service change, these changes must be propagated to all nodes within the smart space, which have installed that service [P.18]

<R.15> Access right revocation and update when node-churn occurs must be guaranteed [P.2] [P.19]

2.9.2 Non-Functional

Non-functional requirements are defined based on the nature of the scenario and considerations regarding performance issues:

<R.16> Minimize network/resource and computational impact caused by issuing local service certificates [P.20]

<R.17> Minimize service down-times caused by missing service certificate [P.2] [P.24]

<R.18> Minimize the complexity presented to the user, while managing the security of the system transparently [P.17]

Chapter 3

Related Works

IoT security has been focus of research especially in the past few years, as it started being seen as a real threat to the public. There are many aspects in the IoT that need protection and it is important to identify existing works that already fulfill the requirements defined in section 2.9.

This chapter will first introduce some IoT management systems researched over the years and compare them with the DS2OS in section 3.1. It will then analyze more in-depth several existing app ecosystems in section 3.2, to point out the main protection mechanisms used in the industry and make an analogy with the DS2OS. Section 3.3 will then focus on security-specific innovative approaches for the IoT, especially certificate-based solutions and token-based authentication.

3.1 IoT Management Systems

IoT management has become increasingly problematic over the years, due to the obstacles analyzed in section 2.1. Several attempts to solve this issue have been documented. No industry standard exists, as of yet, but many possibilities have been researched in the past.

The related works presented below focus on different aspects: the usability, the simplicity, the platforms to integrate third-party providers, the security and so on. The top-level differences between these related works and DS2OS will be briefly analyzed.

ThingStore

ThingStore [1] is a platform to bring different actors of the IoT cyber-physical world together. The focus is on an IoT Market Place, where smart services are available for consumers. Smart services can be dynamically offloaded to edge nodes and interact with one another (or with the platform) via a custom event query language.

While the offloading approach is valid, the platform heavily relies on the cloud infrastructure; This is not the focus of the DS2OS, where smart spaces and applications are decentralized and autonomous.

OpenIoT

Similarly to *ThingStore*, the OpenIoT proposed in [47] focuses on open platforms, accessible by service providers and clients to enable a global IoT ecosystem for everyone. The solution is a centralized platform for developing, managing and accessing applications for the IoT. Users may directly access remote IoT devices by using specific API and web applications created by developers and service providers.

IFTTT

If This Then That (IFTTT) is a web-service mainly made for personal use [35]. It is used to connect and automate actions involving different apps, industrial products and well-known services. It requires ad-hoc service integrations, to allow communication over a defined interface.
The solution allows users to to set up cause-effect triggers that perform a set of pre-defined actions (e. g. if the temperature raises above a certain value, enable the air conditioner).

The solution has a huge potential as it is completely decentralized and enables simplified management for users. Also, it isn't limited to a smart space, but covers the entire gamma of services tied to a user, ranging from social applications to control of smart devices. It is, however, still somewhat limited and devices need to support its APIs in order to function.

HomeOS

Microsoft proposed an OS for the home in [17]. *HomeOS* empowers the non-expert user to manage a home environment, filled with IoT devices connected to the network, over a user-friendly UI.

A centralized application, acts as the controller of the entire home. The heterogeneity of devices is overcome by offering a set of device-specific drivers, which are run on the lower layers of the proposed solution.

The UI presented to the user resembles a traditional Desktop UI. The user interacts with the OS, rather than with the devices and applications themselves. New applications and devices are considered as plug&play modules from the management layer, similarly to a desktop OS.

SAVI-IoT

A valid example of an autonomic management system is given in [45], where IoT is provided as a service. The solution uses a multi-layered architecture: data aggregators accumulate data and communicate over a middleware, while more complex services can be deployed on edge nodes or on the cloud, depending on the performance requirements. Compared to the service management approach chosen for the DS2OS, SAVI-IoT leverages *Docker* and *Swarm* to manage services, both in the cloud and on edge nodes. This is a significant difference, as the management logic and the security mechanisms are automatically taken care of by these existing engines.

3.2 App Ecosystem study

Security approaches in diverse app ecosystems were introduced in section 2.6.

Different App marketplaces exist nowadays in the industry, with different target devices. The three main marketplaces that will be analysed and compared are:

- iOS App Store [42]
- Google Play Store [7]
- Ubuntu Store [81]

These specific marketplaces have been chosen because they are particularly significant for this study and had security-related documentation available. The first two are the biggest and most widely used App stores nowadays and target mobile devices; the last one targets any device running Ubuntu but supports distribution on IoT devices running Ubuntu Core.

3.2.1 Google Play Store

The *Google Play Store* is the marketplace, hosted by Google, used for distribution of apps for the *Android* platform. The Android ecosystem specifically targets mobile devices, such as tablets and smartphones, and is the most widely used mobile platform as of today.

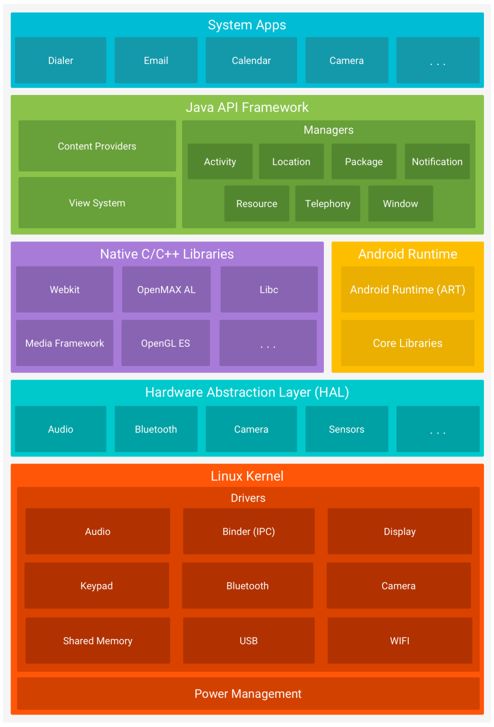
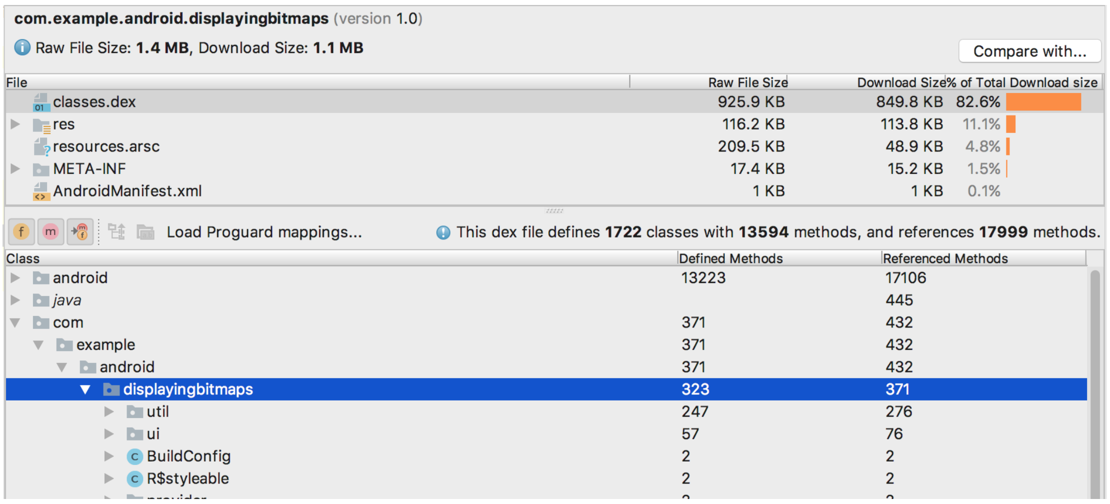


Figure 3.1: Android platform software stack [6]

The mobile platform itself runs on top of a modified *Linux kernel* and mostly open source libraries, as visible from the architecture diagram in 3.1. Developers are required to use the provided Android SDK, which leverages native libraries and can access the device's hardware via the HAL.

Applications are packaged by the developer in `.apk` bundles and contain the elements shown in 3.2:

- compiled classes in a `.dex` format, executable by the Android Runtime;
- `res` folder, containing not compiled images and resources;
- `resources.arsc`, containing compiled resources such as binary XML;
- `META-INF` folder, in which the Java Manifest and the application certificate are stored (more details about the certificate in 2.6.3);



com.example.android.displayingbitmaps (version 1.0)

Raw File Size: **1.4 MB**, Download Size: **1.1 MB**

Compare with...

File	Raw File Size	Download Size	% of Total Download size
classes.dex	925.9 KB	849.8 KB	82.6%
res	116.2 KB	113.8 KB	11.1%
resources.arsc	209.5 KB	48.9 KB	4.8%
META-INF	17.4 KB	15.2 KB	1.5%
AndroidManifest.xml	1 KB	1 KB	0.1%

Load Proguard mappings...

This dex file defines **1722** classes with **13594** methods, and references **17999** methods.

Class	Defined Methods	Referenced Methods
android	13223	17106
java		445
com	371	432
example	371	432
android	371	432
displayingbitmaps	323	371
util	247	276
ui	57	76
BuildConfig	2	2
R$styleable	2	2

Figure 3.2: Contents of an Android PacKage (APK), shown using the APK Analyzer [28]

- `AndroidManifest.xml`, containing metadata such as the App ID, references libraries or access rights;
- `assets` folder, in which additional assets are kept, such as `.html` files;
- `lib` folder, where compiled library dependencies are stored

Application bundles (APK) submitted to the Google Play Store undergo a review process (mostly automated), to verify that the app does not contain malicious code and fulfils all the requirements imposed by the ecosystem.

Published apps can be controlled by developers via a web-based dashboard UI. This feature is very similar to the ones offered by other app stores, and includes update mechanisms, analytics, feedback channels, marketing control and so on.

Code Signing

Each app submitted and distributed to Android devices has a pinned certificate, self-signed by the developer. The certificate contains a checksum for each file found inside the APK. The default digest algorithm is SHA-1.
Code integrity and signature checks are performed when submitting an app to the store and before installing an app downloaded from the store. The checksums contained inside the APK represent the values that need to be verified against. If the checks fail, an error is raised and the installation fails.
After a successful installation, a feature called *Verify Apps* periodically kicks in. It scans the device for potential harmful apps and verifies that all apps are behaving correctly.

It is worth mentioning, that within the Android ecosystem, users have the possibility to install apps from non-trusted sources, i.e. not from the Google Play Store. Integrity verification is performed even when doing this, but there is no real guarantor. The source of the app could be anyone and nobody likely ever scanned the code for identifying threats.

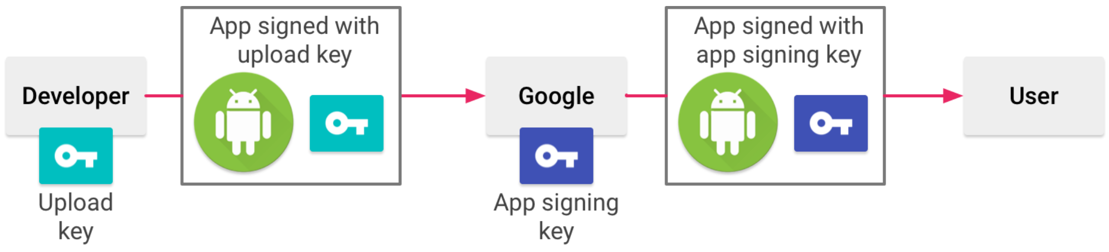


Figure 3.3: Code signing mechanism for Android apps using the signing key stored on Google's servers

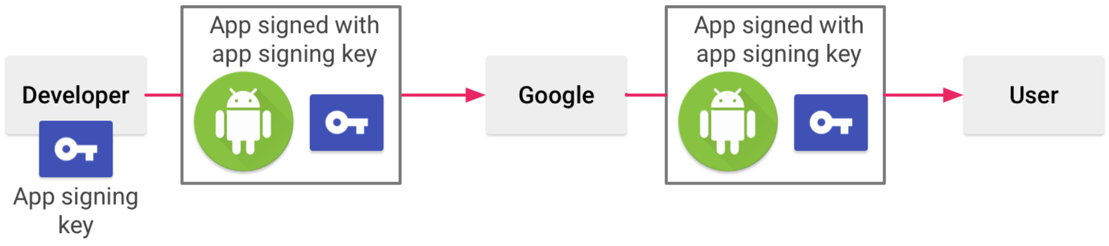

Developer
App signing key
App signed with app signing key
Google
App signed with app signing key
User

Figure 3.4: Code signing mechanism for Android apps relying on developer's key

Google offers two different methods for code signing an application. These are respectively shown in Figure 3.3 and 3.4.
The **first** one relies on the developer uploading the signing key to the Google servers (using a dedicated tool for securely encrypting and transmitting the key). A further key is then created by the developer and registered with Google. The latter is only used for signing the application before it's uploaded, but is then re-signed by Google after the app is verified. The purpose is to keep the real signing key secure, while the developer holds a key only used for uploading to the store. If this key is ever lost or compromised, it can be revoked and a new one generated.
The **second** method rests entirely upon the shoulders of the developer. The developer signs an app directly with their key, which needs to be kept safe. If the key is ever lost, there is no way to restore it and no further updates of the app are possible.

Runtime Environment (RTE)

Apps are run within the Android Runtime (formerly Dalvik Virtual Machine), which translates byte-code to machine instructions. This behavior is very similar to the one provided by the Java Virtual Machine (JVM). Each application is associated to a separate user, given by the unique ID of the app. When launched, the application is run in a virtual sandbox, to keep it from accessing anything outside itself. This approach is a step beyond other existing OS, such as Linux, where multiple applications may be run with same user permissions.

Access rights and dependencies

On Android, apps that require additional functionalities, not available in the sandbox, require the user to set them, in the form of user permissions. This happens either at install time or runtime. To allow an app to request permission for a resource, the required user permissions need to be declared in the manifest. User permissions include privileged access to hardware resources (e.g. microphone, camera, bluetooth and so on), user data or other apps.
It is ultimately up to the user, to decide whether a specific permission should be granted to an app, based on an educated guess. Permissions can be managed from within the system, giving users the possibility to grant or revoke them later on.
Here is an example of the permission declaration for accessing the SMS functionality inside the app manifest:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.test-app">

    <uses-permission android:name="android.permission.SEND_SMS"/>

    <application>
        ...
    </application>
</manifest>
```

Apps can rely on other apps for certain optional functionalities. An example is the media sharing functionality. In order for a camera app to share media content, it redirects the user to an other app (e. g. a social media app), with the needed metadata.
This dependency mechanism is limited and doesn't allow apps to directly access other apps, but rather transfers control over to them. Security-wise, this is a favorable feature, but doesn't provide the flexibility that other systems have.

3.2.2 iOS App Store

The *Apple App Store* is Apple's own marketplace, used for distributing Apps for *iOS* devices. The ecosystem is completely proprietary and strongly revolves around the target physical devices, produced and sold by Apple itself. In general, Apple favours more rigid guidelines, which third-party developers must comply to when building their Apps.

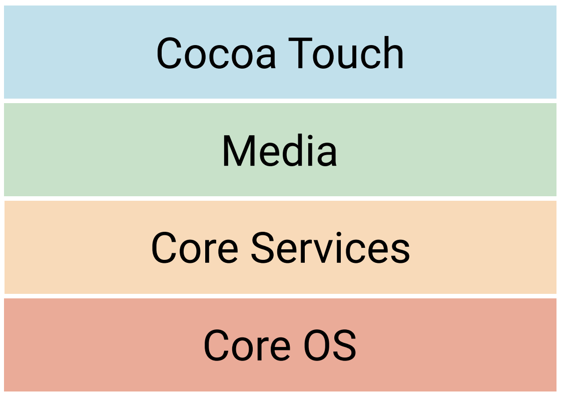


FIGURE 3.5: iOS Software Stack Architecture

The iOS OS itself is Unix-based and only runs on Advanced RISC Machine (ARM)-based mobile devices produced by Apple. The system software stack is in some ways similar to the one used by Google (see 3.2.1). As visible in 3.5, the system services and media libraries (i.e. native C/C++ libraries) run on top of the core OS. Applications primarily rely on the *Cocoa Touch* layer, which drives the UI and provides access to main system functions (touch input, camera and so on). Other services provided by the *Core Services* and *Media* layers are also accessible to some extent. The *Core OS* layer offers Bluetooth and Universal Serial Bus (USB) functionality, but the developer has no direct access to kernel operations.

Similarly to the Android case, within this ecosystem only Apps developed with the iOS SDK are supported. These may only be installed and run on iOS devices.
The App submission to the store, the review process and the store listing management are very similar to what was described in 3.2.1. The review process, however, is typically stricter and with more human intervention.

Code Signing

Before being uploaded to the Apple App Store, an `.ipa` archive is built [36]. The structure of such an archive is similar to the structure of an APK bundle. The bundle contains `iTunesArtwork` image and metadata files, which are used for App listing on the store page, and a `Payload` folder, where all the App data is kept:

- `Info.plist`, an XML file containing essential configuration information for a bundled executable (similar to a manifest file), such as the bundle ID, the version, the required access rights and so on;
- `Executable`, which is the the compiled executable binary;
- `Resource` files, such as assets, layout files and so on;
- Localised assets (for multi-language support);
- Icons for the home screen;

Application packages are code-signed, guaranteeing the integrity and authenticity of apps. The procedure is already shown in Figure 2.12 and involves an Apple-issued certificate. Apple enforces the code-signing mechanism even more than Google does: any executable or library that may be run needs to present a valid signature.
The code signature is validated every time an application is launched. Only validated code can be run on iOS systems, including the system libraries. This feature prevents apps from loading unsigned code or using self-modifying code.

Besides code-signing apps, Apple also enforces code-signing on compatible IoT devices. Vendors of IoT devices must sign their embedded software with an Apple-issued certificate, in order to be compatible with iOS native API. This feature is similar to the one introduced in this thesis, although the considered scenario is different and the entity certifying the application is the Apple App store.

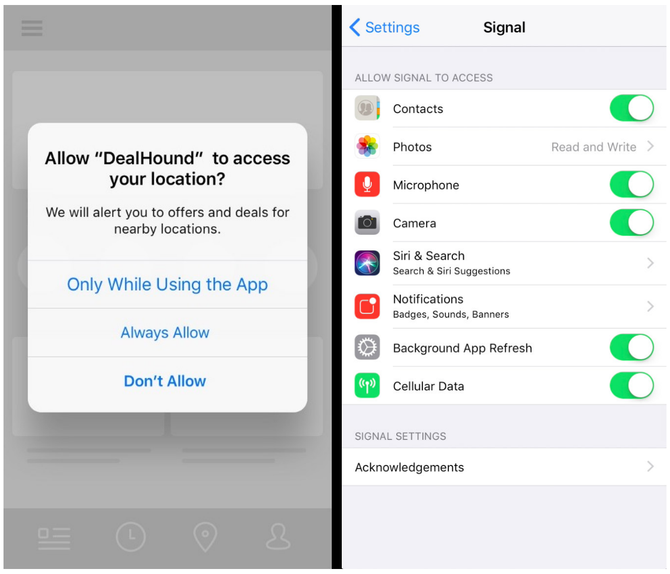
Runtime Environment (RTE)

iOS runs third-party apps within a sandbox. Most apps and iOS software run as "mobile" user. Upon installation, each app is randomly assigned a unique home directory with restricted permissions. This prevents other apps from reading or modifying files stored by other apps.

Access rights and dependencies

iOS enforces access control over some hardware resources and personal data. The latter is taken especially seriously, as it involves personal data and could lead to a privacy violation. Personal data includes contacts, photos, location, calendar information, social media accounts and more. Users can grant, view and revoke permissions to an app,

similarly to the Android case.
Additionally, Apple offers shared data repositories for specific purposes, such as health data or smart home appliances. Apps can share domain-specific data over this interface. The same user permissions apply and are very security-critical.



Allow "DealHound" to access your location?

We will alert you to offers and deals for nearby locations.

Only While Using the App

Always Allow

Don't Allow

Settings — Signal

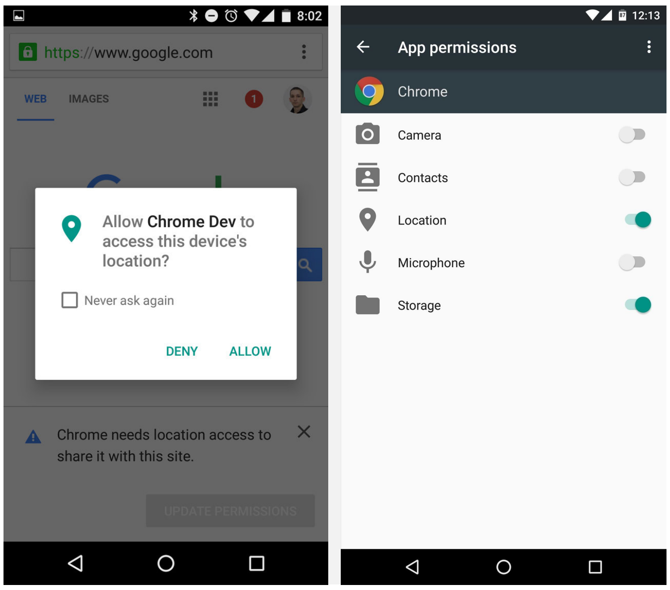
ALLOW SIGNAL TO ACCESS

- Contacts [✓]
- Photos — Read and Write
- Microphone [✓]
- Camera [✓]
- Siri & Search — Search & Siri Suggestions
- Notifications — Badges, Sounds, Banners
- Background App Refresh [✓]
- Cellular Data [✓]

SIGNAL SETTINGS

- Acknowledgements

Figure 3.6: iOS app permission settings; left: a dialog window appearing when the app requires a resource; right: the complete set of user permissions set by the user for a specific app



https://www.google.com

Allow Chrome Dev to access this device's location?

Never ask again

DENY ALLOW

Chrome needs location access to share it with this site.

UPDATE PERMISSIONS

App permissions — Chrome

- Camera [ ]
- Contacts [ ]
- Location [✓]
- Microphone [ ]
- Storage [✓]

Figure 3.7: Android app permission settings; left: a dialog window appearing when the app attempts to access a resource; right: the complete set of user permissions set by the user for a specific app

In figures 3.7 and 3.6, the iOS and the Android approaches for granting access rights to an application are compared. The two models are very similar and provide the same UI controls to the end-user.
The same dependency mechanisms between apps introduced in subsection 3.2.1 are used on iOS.

3.2.3 Ubuntu Store

The last application ecosystem worth mentioning in this analysis is *Ubuntu Core*, developed and maintained by Canonical. Ubuntu Core is actually the name of the host OS on which the applications run. It is, however, strictly coupled with a distribution system called *Snappy* [73], developed by the same company.

Ubuntu Core is a lightweight version of the popular *Linux Ubuntu* distribution, built for IoT devices and with a strong focus on security. It adopts the same kernel as classic Ubuntu, but with a much smaller footprint and fewer features. The resulting OS image size is approximately 350 Megabytes (MB). This may still be too big to be run on micro-controllers, but is fit for small single-board computers, such as the Raspberry Pi.

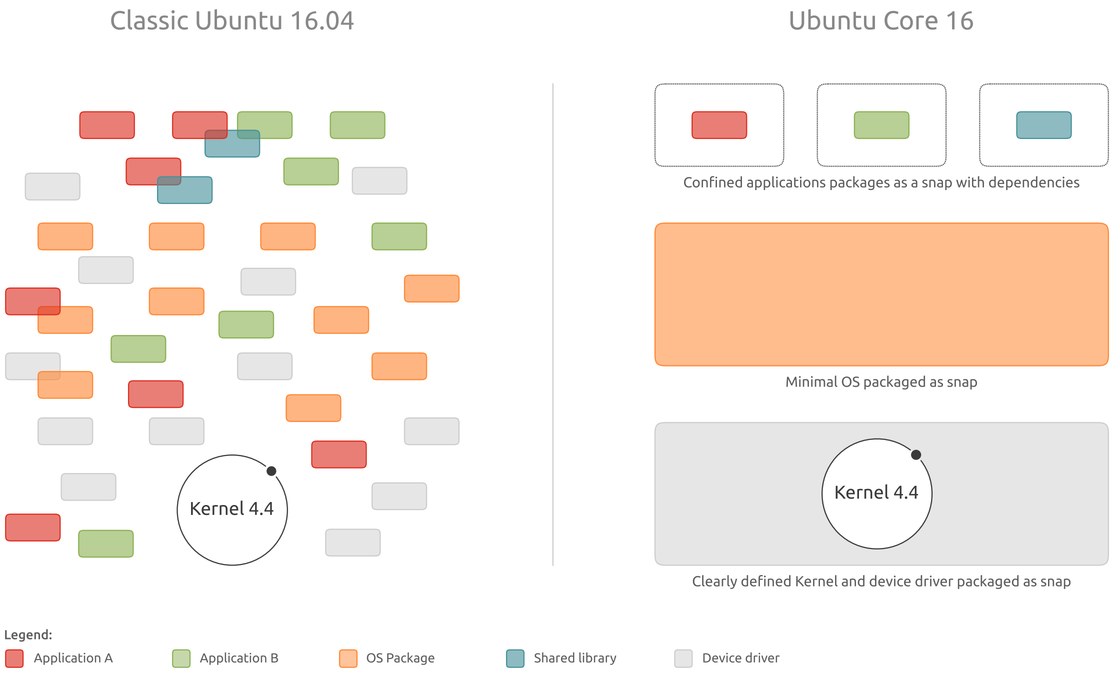

Classic Ubuntu 16.04
Ubuntu Core 16
Kernel 4.4
Confined applications packages as a snap with dependencies
Minimal OS packaged as snap
Kernel 4.4
Clearly defined Kernel and device driver packaged as snap
Legend:
Application A
Application B
OS Package
Shared library
Device driver

Figure 3.8: Comparison between Classic Ubuntu application distribution and Ubuntu Core distribution with Snappy [80]

Snappy is the system responsible for installing and updating applications on devices powered by Ubuntu Core. Snappy is based on *snaps*, which are containerised software packages, saved in the `.snap` format. Snaps are language-independent and self-contained software packages, which run on the target OS as standalone applications.

Every software component running on Ubuntu Core devices must be package as a standalone snap, including the OS and the kernel. Snaps are not submitted to the Ubuntu archive but to a store, where they can be downloaded from. There is very loose coupling between all the snaps, which may also be updated independently of one another, without breaking any dependencies.
Snaps are meant to bundle together everything the app needs, to give the developer better control over the applications' RTE. The data produced and used by the snap is kept separate as well.
Similarly to the other solutions analyzed in 3.2.1 and 3.2.2, submitted snaps undergo an automatic review process. The kernel, OS and gadget snaps require a manual review. The distribution and update mechanisms also works similarly to Android and iOS.

This particular ecosystem is very relevant for the purpose of this thesis and a valid comparison to the DS2OS. This is mainly because both target IoT devices and try to abstract dependencies between application bundles, although using a different method.

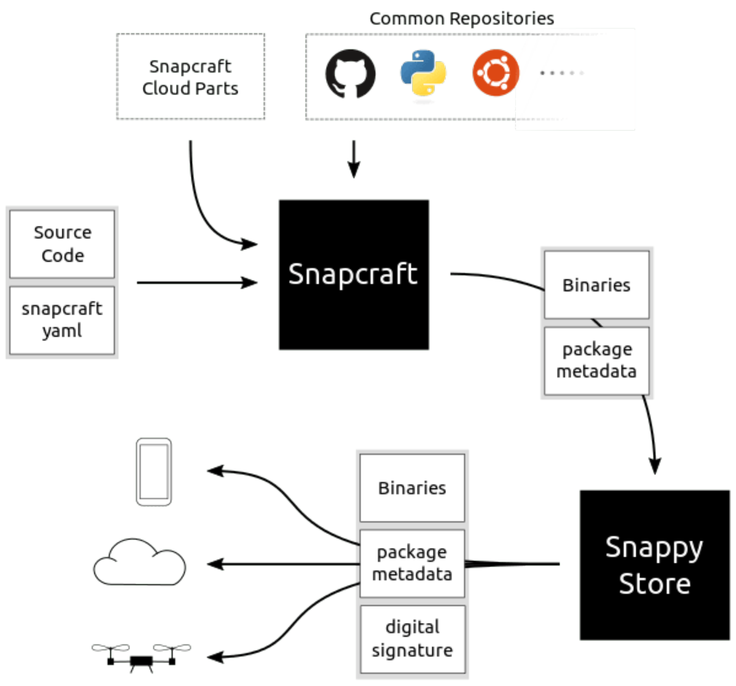


Figure 3.9: Packaging workflow of a snap

Code Signing

In the Ubuntu ecosystem, signed source packages, as well as binaries built from such signed packages, are checksummed using [65], [43] and SHA-256 [32] algorithms. Packages also contain metadata defined by the developer in a `snapcraft.yml` file. Important information, such as hardware and snap dependencies are contained in this file. The checksum is attached to the release file, which is then GPG-signed by Ubuntu [62], before making it available on the store. Snaps are not directly code-signed by the developer but only by the store.
Figure 3.9 shows the entire snap creation, code signing and distribution workflow.

Compared to Google's and Apple's approach, applications on Ubuntu Core are only verified during installation.

Runtime Environment (RTE)

Snaps run in a restrictive security sandbox and have very limited contact with other snaps. The parameters for this sandbox are defined within the snap packaging. This security sandbox is essential, as app snaps are executed with root privileges on Ubuntu Core. Modification of the security sandbox is not possible, while the snap is being run.

Access rights and dependencies

The Snap ecosystem encourages developers to bundle everything they require within their snap. Other required hardware and software resources are treated as external dependencies.

While Android and iOS combine software dependencies management and access rights in one concept, Ubuntu Core uses an interface-based approach to dependency management. This is because interactive access control is harder to achieve in Ubuntu Core, since snaps often run as system services. Dependencies can be accessed via interfaces provided by the OS or other snaps.
System services and other apps can offer an interface, for other apps (clients) to plug in to. Natively supported interfaces are mainly tied to system services such as calendar, bluetooth, networking, firewall, docker and many more. Non-privileged interfaces (e.g. network binding) are automatically hooked up to the application. Interfaces requiring privileged access must explicitly be connected by an administrator.

Interfaces follow a client-server architecture, allowing many snaps to use the functionalities offered by a single interface. To use an interface, developers declare it as required

inside the `snapcraft.yml` file. Multiple interfaces can be used by a single snap. The requirement declaration, together with privileged interfaces management, represent the access rights in the Snappy system.
The approach provides high flexibility, similarly to context models (see subsection 2.4.2), but also increases the snap review overhead.

3.2.4 Comparison

The three analyzed solutions have similar approaches and represent today's standard in app security.
Table 3.1 highlights which of the defined requirements are covered by the analyzed systems.

	Core		
<R.1> Service Integrity on S2S	✓	✓	✓
<R.2> Service Distribution Integrity	✓	✓	✓
<R.3> Installed Service Integrity	✓	✓	✓
<R.4> Site Metadata Integrity			
<R.5> Developer Authentication	✓	✓	✓
<R.6> Download Services certified by S2S	✓	✓	✓
<R.7> Locally Signed Services			
<R.8> Authenticate User-downloaded Services			
<R.9> Node Authentication			
<R.10> Periodic Node Identity Verification			
<R.11> Access Groups		✓	✓
<R.12> Interactive Access Rights settings		✓	✓
<R.13> Access Rights unique for a Service	✓	✓	✓
<R.14> Propagate Access Rights changes	✓	✓	✓
<R.15> Support Node-churn			
<R.16> Minimize resource impact	✓	✓	✓
<R.17> Minimize service down-times	✓	✓	✓
<R.18> Minimize UX Complexity		✓	✓

Table 3.1: Comparison between requirements covered by existing app management system. From left to right: Ubuntu Core, Android, iOS

The main difference between the compared solutions and the DS2OS lies in the deployment target. The DS2OS targets unattended distributed nodes and therefore adds an orchestrator entity between the store and the nodes.
Even though Ubuntu Core targets IoT devices, the app installation and update mechanisms still reside on each target device. Communication with the store is direct and doesn't require an intermediary. While this approach is simpler and doesn't require additional security mechanisms, the advantages of having a distributed system were already presented in 2.4.

While the analyzed stores provide an optimal level of service security, leveraging code-signing, they fail to address a distributed scenario.

It is worth mentioning, that the file system of both Android and iOS devices is typically encrypted by default. This additional feature prevents attackers from tampering with the system by simply gaining physical access to the device.

3.3 Approaches to IoT Service Security

In the analysis, two main approaches for guaranteeing authenticity of services and safeguarding application access rights were identified: digital certificates and authentication tokens.
In both cases, the presence of a dedicated authentication server is a key component. Be it a CA or an authorization server, this entity is fundamental. It is required both in a local and in a global scenario, because it shifts the burden of authenticating devices and services to the trusted entity.

Most of the works analyzed below adopt some authentication server, but with a different scope than in the DS2OS. The scope of the authentication is typically either a session or a connection between two devices. The goal is two simply allow to nodes to securely communicate with one another. The DS2OS, instead, aims to authenticate single services, making the solution more granular.

In [46], a scenario similar to the DS2OS is considered. Devices and sensor nodes have intermittent connectivity. The solution introduces several software entities, which are deployed within an IoT network and allow to manage the communication between the nodes.

Secure communication is achieved by using *session keys*. Keys are distributed by a dedicated entity, the *Auth* server. The server has the task of authenticating and authorizing locally registered entities (i. e., nodes). This fulfills <R.9>. In comparison, the VSL leverages directly TLS connections (see subsection 2.4.2), to generate session keys.

To obtain authorization in [46], an entity needs to have a valid distribution key, set up during the registration with the Auth server. If a device is constrained, this distribution key can be a pre-shared key, otherwise traditional PKI is used.

Like in the DS2OS, the basic assumption is that the Auth entity is under control of the user and can be trusted. Compared to the DS2OS, the solution supports multiple authentication entities for improved scalability.

While communication between two entities within a smart space is secured, the integrity of the applications running on them is not handled.

3.3.1 Certificate-based Security

Several works leverage digital certificates for creating secure communication channels between the nodes. Among the research targeting specifically IoT nodes, some works are worth mentioning.

Panwar et al. propose the usage of COAP over a Datagram Transport Layer Secure (DTLS) channel in [60]. Compared to traditional TLS, this requires less hardware resources and may be more appropriate for IoT devices. Instead of pre-shared keys, traditional PKI is used in this case. A dedicated CA, considered to be the trusted entity, allows nodes in the network to request a certificate. <R.4>, <R.9> and <R.16> are tackled, but no application-specific security is considered.

The possibility to store access control rules and other metadata inside certificates is introduced in **Schukat** [69]. The authors suggest that a PKI-based certificate solution for the IoT will become a cornerstone for a more secure infrastructure. However, they also analyze several problems:

- identifying IoT devices globally and using a valid domain name for them within a certificate is difficult (IPv6 may be a solution)
- certificates are large and require hardware capable of performing cryptographic operations
- theft of intermediate certificates is a dealbreaker when billions of IoT devices affected
- there is no need to globally validate every IoT device. Therefore using a single tier CA hierarchy can be an advantage
- devices require a pinned certificate before deployment, in order to request further ones
- proper management is required to sign and replace certificates on a large number of devices

Some of these issues do not focus on services, but on IoT devices as a whole. This thesis aims to go beyond the simple device authentication and focus on securing applications. Most of the analyzed considerations are valid and can be considered during the design phase, but do not cover directly the functional requirements defined in section 2.9.

In the mentioned works, the use of pre-shared keys for authentication on IoT devices is critized. It is argued that pre-shared keys are an insecure solution: the source of a key is not guaranteed and they may be compromised at any time, without a backup plan. Having a solution in place, that goes beyond static keys, makes the entire system dynamic and more secure.

Short-lived Certificates

The possibility to overcome certificate revocation techniques, such as CRL and OCSP, has been discussed several years ago in [66] and [50].

Topalovic presents an approach [79], based on the two above mentioned works, that implements this concept. Requirements <R.10>, <R.14> and <R.15> are tackled in this work.
The key idea is to add security to the certification system, by giving the power back to the CAs. Certificates in this solution have short lifetimes and compliment Google Chrome's CRL. Combining the two techniques, the authors claim their solution to be more secure than OCSP, while reducing the setup time of TLS connections. It is also stressed that certificates should be renewed shortly before the expiration date, to have some safety buffer for the renewal process. This concept is re-introduced as renewal window in subsection 4.2.5.

Performance Impact

Approaches based on PKI certificates have one major downside: the size and overhead can be considerable. A certificate can be up to 2 KB large [69], and handling multiple certificates of that size can be tough on IoT devices.

To overcome this problem, Park [61] introduces the idea of using *implicit certificates* to significantly reduce the size of a certificate. This solution is particularly interesting, as it addresses <R.16> and the certificate renewal described in subsection 4.2.5 could benefit from it.

3.3.2 Token-based Security

In subsection 2.8.3, the usage of authentication tokens for the IoT was theorized. These are mainly used for access control and authentication, but are a valid and light-weight alternative compared to attaching access rules to digital certificates (see subsection 2.4.10).

ACE is a solution proposed in [9] by the Internet Engineering Task Force (IETF), that utilizes Proof-of-Possession (PoP) tokens for constrained environments. Compared to traditional token systems (see subsection 2.8.3, this solution targets not only HTTP over TLS, but also COAP. The proposed approach leverages OAuth and covers <R.9>, <R.13> and <R.15>.

The *OAuth-IoT* solution proposed in [70] is based on the efforts of the IETF and aims to improve upon the foundations set in [9]. The solution integrates an IoT gateway, which acts as an authorization server and routing gateway, to forward resource requests to the correct IoT node. Harmonizing existing standards together, the OAuth-IoT framework relies on a single centralized entity for authentication and TLS management. The current solution does not tackle P2P communication of IoT nodes, while focusing only on external clients communicating with individual nodes through a gateway.

Another very valid solution for the IoT is proposed in [10]. Bhawiyuga proposes a lightweight solution based on MQTT with authentication. JWT tokens are used for authenticating the IoT devices. A dedicated authorization server entity is required for this purpose. The solution has a different approach from the DS2OS, as it only uses publish/subscribe mechanism to propagate data across a smart space. Also, the used technologies target significantly constrained devices. Similarly to previously discussed works, the proposed mechanisms target <R.4>, <R.9> and <R.16>, but no mechanisms are considered for protecting the applications and managing the lifecycle of the services. Other token-based approaches leveraging the OAuth protocol were considered in [48] and [18].

3.3.3 Comparison

To the best of my knowledge, no other existing works propose an approach based on code-signing applications for distributed IoT nodes, which also leverages a dedicated authorization server. Many works approach the network layer threats, introducing authorization server to ultimately enable an efficient creation of secure channels between IoT nodes. However, service-specific protection is not considered. The solutions analyzed in section 3.2, instead, focus entirely focused on app management, but target a different scenario.

Among the analyzed works especially [60] and [61] are meaningful for future works, as the suggested solutions could allow to decrease the overhead of the approach proposed in chapter 4.

Chapter 4

Design

The requirements formulated in section 2.9 need to be fulfilled in order to protect μ-services in smart spaces and ultimately prove the proposed thesis. This chapter focuses on the design of the solution, target of this work.

The chapter is structured as follows.
The specific protection mechanisms for each μ-service are introduced in section 4.1. These are put into context in section 4.2, where the automated certificate management approach is described.
An overview of how the introduced security mechanisms are used to protect service management operations is given in section 4.3. The setup of the security features within a distributed system is presented in section 4.4.

4.1 μ-service Security

The functional requirements analyzed in section 2.9 focus on *integrity*, *authentication* and *access control* security goals. To fulfill these requirements, additional security mechanisms for the DS2OS are proposed.

The DS2OS relies mainly on μ-services and high-level service management. Every component within the DS2OS runs as a μ-service, with the exception of the VSL and the SHE. For this reason, the security mechanisms described in this section focus on securing μ-services and protect the service management features of the DS2OS.

4.1.1 μ-services in the DS2OS

μ-services in the current DS2OS implementation are simple *OSGi* [2] services. They contain library dependencies (if any) and can access interfaces offered by other services according to the configured access rights. However, the service executables do not provide any kind of security mechanism by themselves, except the ones defined for the internal business logic.

As for any other app ecosystem analyzed in section 3.2, application binaries must be wrapped in a package, along with certificates, potential dependencies and other metadata. This is necessary to enforce the protection of the service itself. The app store and the target device require a well-defined package format to process the service.

The same reasoning applies to the adopted OSGi services, which must be packaged accordingly in order to be run within a smart space.

4.1.2 DS2OS Service Package

A service package in the DS2OS consists of:

- the OSGi **executable** of the service
- the **context model** of the specific μ-service (see subsection 2.4.2)
- the DS2OS **manifest** of the service
- a **service certificate** signed by the developer

The contents of the package are strictly kept in the base folder, with no sub-folders or internal hierarchical structures.

Packages are compressed in ZIP format and saved with `.zip` extension.

To package a service, developers can leverage a dedicated tool: `packager.jar`. The packager tool takes in the executable, a manifest defined by the developer and the context model. Then it automatically generates a service certificate with the signature of the developer and creates the service package.

Service Executable

The service executable is a `.jar` archive, containing generated Java bytecode. It contains a `MANIFEST` file, with metadata required by OSGi. All library dependencies are already contained within the archive.

The executable must be able to run on a standalone machine without any ulterior library dependencies.

Context Model

The context model is defined by the developer in an XML file and is unique for a μ-service. A context model must contain the declaration of the service context node and all its children, along with their access control rules.

In the current implementation, the S2Store does not perform validation on the context model, but this is planned for future works.

Service Manifest

The service manifest is essential for a service to be recognized as a valid DS2OS μ-service. It contains:

- the unique ID of the service
- the ID of the developer
- the version of the service
- hardware resource requirements, such as sensors, CPU or storage constraints
- the required context models **(optional)**
- the conflicting context models **(optional)**
- the list of access rights **(optional)**
- the cryptographic hash of the service executable
- the cryptographic hash of the service context model

A service manifest is not to be confused with the traditional Java `MANIFEST.MF`, which is still contained within the executable archive. A service manifest is defined in JSON format and must contain at least the fields described as non optional. An example of a valid DS2OS manifest for a temperature service is shown in listing 3.

The **service ID** allows to uniquely identify a service from the development stage all the way to the deployment on an edge node. To identify each service, its **Java package name** is used. This approach is borrowed from the Android ecosystem.

```
{
        "serviceId": "com.bosch.sensor.bme280.temperature",
        "developerId": "0000042",
        "versionNumber": "1.0",
        "hardwareResources": [
                { "type": "sensor", "value": "BME280" }
        ],
        "requiredContextModels": [],
        "conflictingContextModels": [],
        "accessRights": [],
        "executableHash":
        ↪  "516d0cec88c7fc35c17f5accbc3a76aa5e3c029aa3976cf4dce11a7560042f67",
        "contextModelHash":
        ↪  "d68a8f6e846bed651b634dbce9fffee4e0bd5684e5463344bdcbf92b3af30bcd",
}
```

Listing 3: DS2OS Manifest for a Temperature Service

The **developer ID** is provided by the S2Store, after the developer registered his profile account. The registration process may involve an additional verification process in the future, to ensure the identity of the developer.

The **cryptographic** hashes over the full service executable and context model are generated with the SHA256 algorithm.

The required and conflicting **context models** represent the dependencies to other μ-services. A context model is identified by the unique ID of the service it is defined for.

The list of **access rights** contains the declaration of access rights, which are needed to run the service (or certain features). Access rights are described in detail in subsection 4.1.4.

A service manifest is defined by the developer and cannot be changed by any other entity afterwards. Only the developer may apply changes in later versions of the service.

Service Certificate

The certificate pinned to a service within a package is one of the cornerstones of the solution.

There are different types of signature, discussed in subsection 4.1.3. The certificate contained within the service package is signed by the developer.

Once verified by the S2Store, a certificate signed by the store itself is added to the same package. For more details about certificate management refer to section 4.2.

4.1.3 Signatures

Three different types of signatures are considered within the DS2OS. Each signature is used to generate a different digital certificate, fulfilling its own purpose. Each signing entity uses a different key for signing a certificate.

The three different types of certificates and signatures considered within the DS2OS are visible in Figure 4.1. In step 1, a developer certificate is generated. Step 2 considers a new certificate, issued by the S2Store, while the certificate needed to run a service within a site is obtained in step 5. The steps are described in more detail in section 4.3.

Developer Certificate

Developer certificates are used to code-sign a μ-service. A certificate is self-signed by the developer for a specific μ-service, in order for it to be published on the S2Store.

The code signing process has two goals:

- providing *authenticity* and *non-repudiation*, ensuring the service really comes from a certain developer <R.5>
- protecting the data submitted to the S2Store <R.1>

The *authenticity* is automatically guaranteed, if the S2Store can verify the identity of the developer.

The *integrity* is protected by attaching a cryptographic hash of the service manifest to the certificate. This certifies that no element within the service package has been modified by third parties from the moment the developer created the certificate onwards.

The exact structure of a certificate is described in section 4.2.

Store Certificate

Once a service has been uploaded to the S2Store, a new certificate is created and added alongside the developer's certificate within the service package.

A store certificate contains the same SHA256 hash over the service manifest, contained in the original developer's manifest. The purpose of a store certificate is to guarantee the trustworthiness of the service, after being reviewed and approved by the S2Store.

This fulfills <R.6>. Furthermore, such a solution is extensible and allows the store to add more metadata to a service certificate in the future.

The store certificate is signed with the store's private key. For the proposed solution, the store is considered a trusted CA and can therefore self-sign certificates. This approach is also more practical for testing purposes. The key-pair of the S2Store is generated once and remains fixed. The public key of the store is the trust anchor, which is known to all smart spaces.

In a future version, the store may just act as an intermediate CA, with the root certificate being issued by a trusted internet CA.

Site Certificate

Certificates must also be issued within smart spaces, to allow attaching additional metadata to a service. When a service is downloaded to a site, a new certificate is issued within the site.

To be run within the site, only certificates issued by the SLCA are valid.

This approach serves several purposes:

- it allows site local metadata to be attached to a service <R.4>
- provides local service authentication, restricted to the user of the site; i. e., a local service with specific access rights cannot be migrated to another smart space <R.7> <R.8>
- offloads the S2Store from the burden of releasing certificates every time a service is deployed

A site certificate is signed with the site's private key. The key-pair is generated within the DS2OS site. The entity in charge of managing the keys and issuing local certificates is a dedicated SLCA.

Every service running inside a DS2OS site must have a valid site certificate. This includes the SLSM, the NLSM, the SHE, the SLCA itself and any other service downloaded from the S2Store.

The specific privileges granted to each service are defined by their access rights.

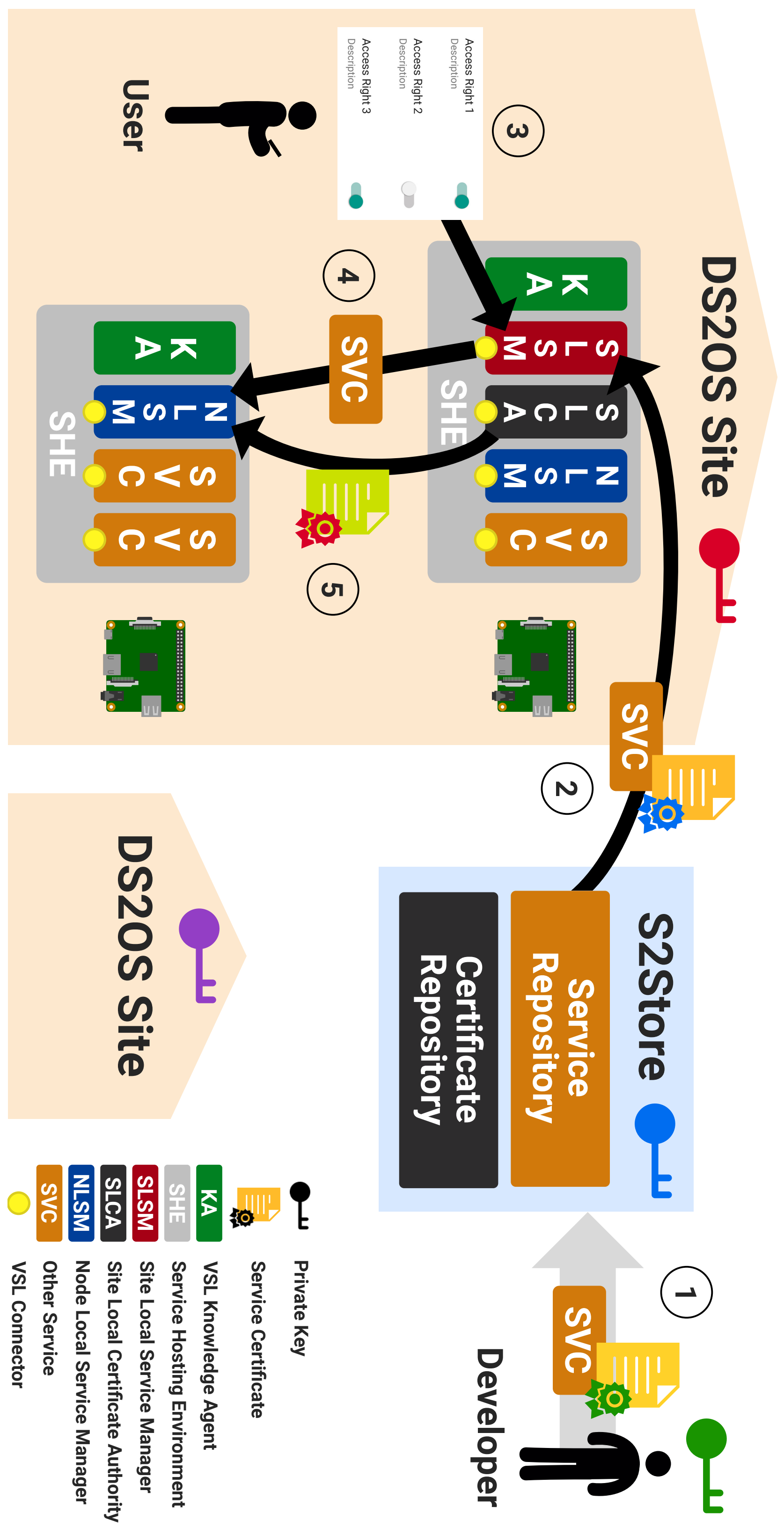


FIGURE 4.1: Security mechanisms in the deployment process of a μ-service in the DS2OS

4.1.4 Access Control

Access control rules make sure, that the data created by a μ-service can only be accessed by authorized entities. In the case of the DS2OS, access rights allow to control access to μ-service context nodes. These may only be visible/accessible to other services, if declared so by the developer and if the correct access rights are set.

Access Groups

Developers declare access groups for their own μ-services, by setting the `reader` and `writer` properties of a context node (see listing 2). Multiple comma-separated access groups can be defined for each context node. The context node is accessible to every service which is part of at least one of the declared access groups.

For example, take a service that can read the temperature from a sensor. The service offers a context node called `currentTemperature`. The definition below allows the `currentTemperature` of that service to be read only by services which are part of either the *home* and/or the *temperature* group.

```
<currentTemperature type="basic/composed" reader="home,temperature"
↪  writer="" />
```

To make a node only accessible to the service itself, the fields must be left empty. In the example above, `currentTemperature` values can be set only by the temperature service itself.

If a field is not specified, the context node will be accessible by anyone.

Access Group Repository

A feature of the DS2OS, is that access groups may be re-used by other developers. Once a μ-service has been submitted to the S2Store, the declared access groups become part of a global *access group repository*. This repository is public and other developers can freely use existing access group names for their own services.

This enforces re-usability of access groups over different μ-services. The targeted requirement is <R.11>.

For example, a temperature service and an air quality monitoring service could both support the *home* access group. A further service could gain access to both these services by declaring only one access right.

Access Rights

When submitting a service, developers embed a list of supported access rights into the service manifest. As shown in listing 3, the `accessRights` values contains this list. Each access right declared in the manifest follows the format defined in listing 4.

```
{
        "name": "temperature",
        "description": "Allows to read the temperature from various
        ↪  sensors in the room",
        "required": true,
}
```

Listing 4: Access Right format defined in a DS2OS service manifest

The name of a supported access right must match the name of an access group contained within the *Access Group Repository* (including the current service itself). The S2Store validates the existence of the declared access groups.

The list of declared access rights must contain the comprehensive list of all access groups, that the service may require. *Optional access rights* may be declared as well.

Upon installation of a service, these supported access rights will be displayed to the user via a UI. An sketch of the proposed UI is shown in Figure 4.2. The user ultimately chooses, which access rights to set and which not <R.12>.

The selected access rights are saved within the VSL of the local site. The access rights are read-only and can only be modified by the user over a suitable application <R.13>.

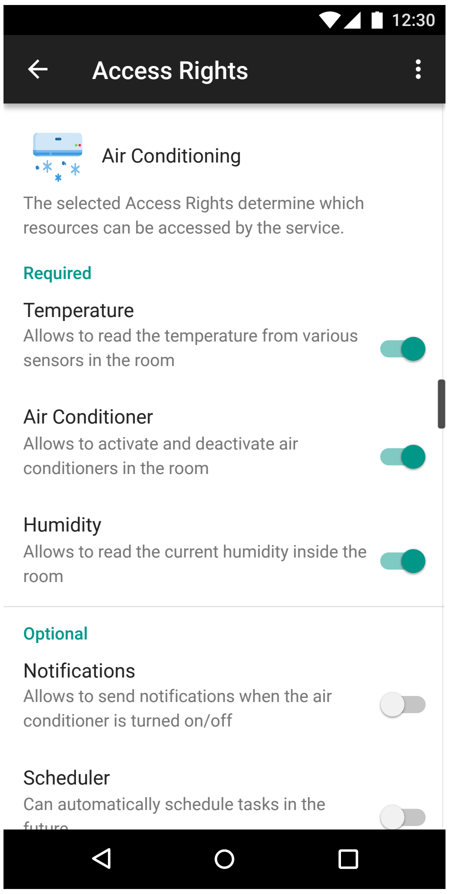
When the downloaded service is finally deployed on a node, the access rights will be read from the VSL and applied to the service. This happens by adding the set access rights as metadata to a site certificate (see subsection 4.1.3). Such a mechanism ensures, that the access rights are integrity-protected <R.4>.

4.2 Certificate Management

The approach chosen for this thesis involves using digital certificates for fulfilling the security goals mentioned subsection 2.5.3.

The reasons for this choice are the following:

- the underlying VSL middleware already provides many security features and is based on PKI certificates


12:30

Access Rights

Air Conditioning

The selected Access Rights determine which resources can be accessed by the service.

Required

Temperature
Allows to read the temperature from various sensors in the room

Air Conditioner
Allows to activate and deactivate air conditioners in the room

Humidity
Allows to read the current humidity inside the room

Optional

Notifications
Allows to send notifications when the air conditioner is turned on/off

Scheduler
Can automatically schedule tasks in the future

Figure 4.2: UI concept presented to the user to manage the access rights of a service

- PKI certificates are a standard, widely used and documented
- using one security mechanism to fulfill several security goals keeps the complexity of the solution to a minimum

While digital certificates are more heavy-weight and not widely used in the IoT, considering the increasing hardware capabilities, this approach is farsighted. Moreover, by overcoming basic hardware limitations, it would be possible to reconcile traditional computing and computing on IoT nodes.

For this work, *X.509v3* [20] certificates are used. X.509 is a standard PKI digital certification system format, first introduced in 1988 and used by all major organizations and governments ever since. Also, X.509 version 3 certificates support attaching arbitrary metadata to them, in the form of extensions. Extensions in X.509v3 are integrity-protected and authenticated, providing the required functionality for this work.

4.2.1 Certificate Authority

In the DS2OS concept, three types of entity can issue certificates: the developer, the S2Store and the Site Local Certificate Authority (SLCA), introduced in subsection 2.4.8.

Developers self-sign their services, without requiring the signature of a public CA.

The S2Store plays the role of a verified CA. It owns either a publicly acknowledged root CA, or an intermediary code-signing certificate, issued by another public CA.

This SLCA is a fundamental component of the DS2OS and is unique for every smart space. The current solution does not support multiple CA entities within the same smart space. This would be feasible, but with an additional management cost. Conflicts could arise, e.g. due to a service obtaining two valid certificates at once and more synchronization mechanisms would be required.

The SLCA has an intermediary code-signing certificate, issued by the S2Store. This certificate is issued when the smart space is first created and stored on the node running the SLCA.

4.2.2 Certificate Structure

As mentioned in subsection 4.1.3, the system considers three different types of certificate. All of them have the same format but are signed by different entities and support different metadata.
The example shown in listing 10 considers a certificate issued by a SLCA for a temperature service, meant to be run within the same local site.

The important fields in a service certificate are: the *issuer*, the *validity*, the *subject* and the embedded *extensions*.

Issuer

This field contains information about the CA that issued the certificate.
The issuer may be one of the entities mentioned in subsection 4.2.1.

Validity

Contains information about the start and end of the validity period of the certificate. The validity start period is always the moment in which the certificate was issued.

The expiration date depends on the service the certificate was issued for, and on the currently adopted policy. For example the certificate of a KA has longer validity than the certificate of a service.

More details about certificate lifetimes are presented in subsection 4.2.5.

Subject

The subject field contains information about the domain of the service. This information includes *common name*, *organization* and so on.

When the certificate is self-signed by the developer, the subject field typically has the same contents as the issuer field.

Extensions

All issued certificates must include some common extensions, plus some optional ones.

A comprehensive list of the currently supported extensions is shown in Table 4.1. The third column shows, whether a constraint is required in every DS2OS certificate.

The extensions *keyUsage*, *extKeyUsage* and *basicConstraints* act together to define the purposes for which the certificate is intended to be used [40]. This mechanism is used to avoid service certificates to be used for issuing other certificates within the network.

The custom-defined extensions for this thesis are *ds2osIsKa*, *ds2osServiceManifest* and *ds2osAccessIds*.

The Object Identifier (OID) used for custom DS2OS extensions are currently not registered on any global authority (such as [34]). For this reason, the OIDs are self-assigned as experimental under the OID `1.3.6.1.3.18`.

Further custom extensions can be defined in the future, using the same OID prefix.

Every certificate has a specific purpose, with specific extensions set.

A **Developer certificate** has the following extensions to define its purpose:

- *basicConstraints*: not a CA, as it is a self-signed certificate
- *keyUsage*: digitalSignature, nonRepudiation, keyEncipherment
- *extKeyUsage*: code signing

OID	**Name**	**R**	**Description**
2.5.29.19	basicConstraints	Y	**Can be critical**. The CA flag must be true if the certificate is issued by a CA
2.5.29.14	subjectKeyIdentifier	Y	**Non-critical**. Identifies the certified public key(s)
2.5.29.35	authorityKeyIdentifier	Y	**Non-critical**. Identifies the public key corresponding to the private key used to sign the certificate
2.5.29.15	keyUsage	Y	**Can be critical**. Contains a bitmask defining the purposes of the certificate: • encipherOnly = 1 • cRLSign = 2 • keyCertSign = 4 • keyAgreement = 8 • dataEncipherment = 16 • keyEncipherment = 32 • nonRepudiation = 64 • digitalSignature = 128 • decipherOnly = 32768
2.5.29.37	extKeyUsage	Y	**Can be critical**. Indicates the purposes of the public key. Extended Key usages defined by [63] are: • Server authentication • Client authentication • Code signing • Email • Timestamping • OCSP signing
2.16.840.1. 113730.1	netscapeCertType	N	**Non-critical**. Old extension used instead of *keyUsage* for compatibility reasons [77].
2.16.840.1. 113730.1.13	netscapeComment	N	**Non-critical**. Additional comment to display on the certificate.
1.3.6.1.3.18.0	ds2osIsKa	N	**Non-critical**. Is `TRUE` if the certificate is meant to be used by a KA, `FALSE` if not.
1.3.6.1.3.18.1	ds2osServiceManifest	Y	**Non-critical**. Contains the SHA256 hash over the service manifest.
1.3.6.1.3.18.2	ds2osAccessIds	N	**Non-critical**. Contains a list of comma-separated access IDs for the service.

Table 4.1: X.509v3 Extensions supported by the DS2OS

Store certificates have the same purpose as developer certificates, except they use the store's signature and have the *basicConstraints* CA value set to true. In the future, other extensions may be used as well.

Site certificates all have the same purpose:

- *basicConstraints*: not a CA
- *keyUsage*: digitalSignature, nonRepudiation, keyEncipherment
- *extKeyUsage*: Client Authentication

KAs are not considered as μ-services, but they also require a valid certificate to run. The certificate must be issued by the same SLCA and contains the following extensions:

- *basicConstraints*: not a CA
- *keyUsage*: digitalSignature, nonRepudiation, keyEncipherment
- *extKeyUsage*: Server Authentication, Client Authentication

Compared to the other described certificates, site certificates require the *ds2osIsKa* and the *ds2osAccessIds* extensions. The *ds2osIsKa* is set to true only for KA certificates. The *ds2osAccessIds* contains the access rights specified by the user for that particular service. The SLCA issuing the certificate checks the VSL for the correct access rights to set for a service (see subsection 4.1.4).

All three types of certificates must always include the *ds2osServiceManifest* extension, which contains the hash of the service manifest. This feature is essential and fulfills all integrity requirements <R.1>, <R.2>, <R.3> and <R.4>.

4.2.3 Certificate Request

Every time a new certificate is needed, it must be requested from the SLCA.
The only entity actively requesting certificates is the NLSM, in charge of managing the lifecycle of services on each node. The workflow is shown in Figure 4.3.

When a new certificate is needed, the NLSM creates a Certificate Signing Request (CSR) [27]. The CSR is sent to the SLSM.
The SLSM performs a preliminary check on the CSR, to establish whether the request is valid within the site. This involves:

- checking the origin of the request, identified by the embedded **public key** of the requester. If the origin is not known, the CSR is denied <R.9>

- lookup the service ID given by the **common name**. If a service with such an ID is not installed in the smart space, the request is considered invalid <R.8>

A valid CSR is forwarded to the SLCA. The SLCA looks up the access rights set for the service, then generates a new certificate for the CSR. The generated certificate is sent back to the requester. A copy of the same certificate is stored locally by the SLCA, for accountability purposes.
If a CSR was deemed invalid, it is denied and an error is sent back to the requester.

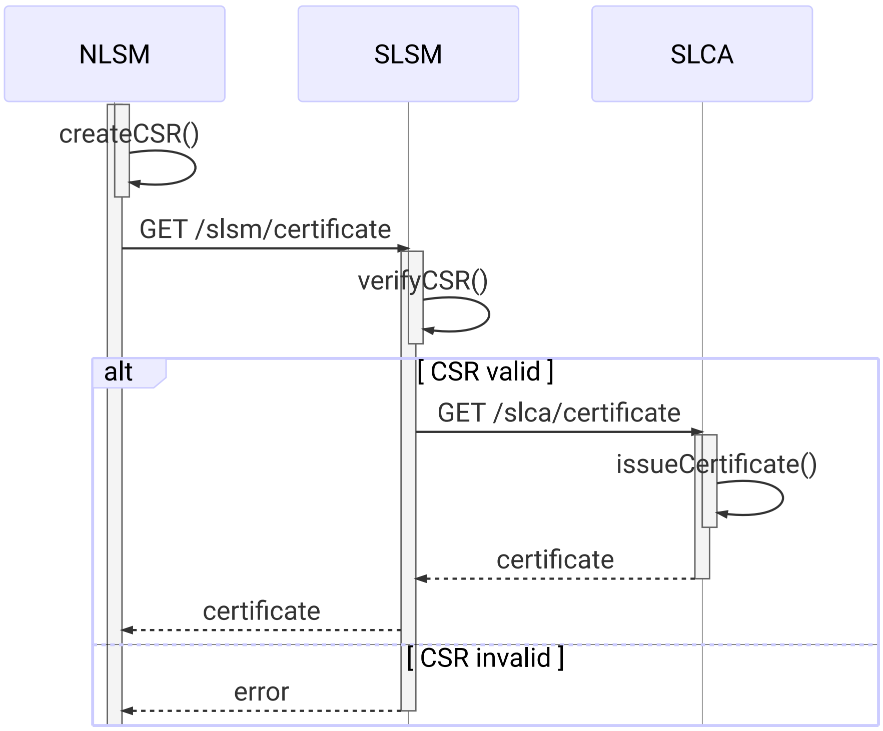


Figure 4.3: Sequence diagram of a certificate request

Splitting the functionality between the SLSM and SLCA is a pure design decision. The purpose is to separate the logic between the two components, so that the SLSM can check the identity of a node and of a service, while the SLCA takes care of creating a certificate with the correct access rights.

The choice of using CSRs, instead of requesting static certificates from the SLCA, has two reasons:

1. the standard PKI approach is a cleaner design solution and preferred, as it allows to verify the identify of the requester via the public key <R.9> <R.10>
2. a CSR can contain additional metadata, making the solution extensible by design.

4.2.4 Certificate Lifetime

Service certificate lifetimes are given by their expiration date.

The idea, proposed in this thesis, is the usage of short-lived certificates. This serves multiple purposes:

- the abolition of more complex mechanisms for certificate revocation, such as CRL or OCSP;
- it ensures that changes to access rights will be propagated to every node in the network within a chosen timeframe, even in the case of node churn <R.15>
- the security of the system is improved, as certificates are always fresh <R.10>

Having a validity limited in time, services periodically require a new certificate. The NLSM is the entity in charge of requesting new certificates before they expire.

Whenever a new certificate is issued, the service needs to be restarted. A restart of the service is needed, in order to execute the integrity checks and register the service with the KA again <R.3>.

The restart process causes service downtime. Management-specific services (e. g. SLSM) require higher up-time and cannot be restarted too often. For this reason, different certificate lifetimes are used. This approach addresses <R.17>.

Service	**Required Lifetime**	**Target Timespan**
SHE	Long	days - months
SLSM	Long	days - months
SLCA	Long	days - months
NLSM	Medium - Long	days
Any other service	Short	minutes - hours

Table 4.2: Target certificate lifetime by service type

The target lifetimes, shown in Table 4.2, are arbitrary and will be the focus of the evaluation in chapter 6.

4.2.5 Automated Certificate Renewal

The core idea is to combine the refresh mechanism used with *authentication tokens*, together with the access rights management introduced in app ecosystems, and embed them into digital certificates. Certificates offer the same advantages that a token approach does, with the disadvantage of being a more heavyweight solution.

The resource control is as powerful as a token-based approach, since the amount of metadata that can be attached to a certificate is virtually unlimited.

Periodic certificate renewals require a certain degree of automation.

The designed solution considers a certificate renewal to be a normal certificate request. Therefore, a renewal follows the same basic procedure described in subsection 4.2.3.

An NLSM is set up to periodically check the certificate expiration date of all the currently running services. If a certificate is about to expire (or if it has already), a new CSR for that specific service is sent to the SLSM.

Different policies are applied to certificate renewal. A **renewal window** constraint is enforced by the SLCA and can be used by every NLSM. Other policies, applied by the SLCA, affect the lifetime of a certificate: **fixed lifetimes** and **random backoff** mechanism. The SLCA can be instructed to prefer one policy over the other by the SLSM.

Renewal Window

A new certificate may not be requested at any given time, but only within a *renewal window.* This window starts at:

$$window_{start} = time_{expiration} - lifetime_{tot} * 0.1$$

The purpose of this time window is to allow an NLSM to fetch a new certificate before it actually expires. This has two benefits:

- it allows to reduce the downtime of the service to a minimum (given by its restart time), addressing <R.17>
- empowers an NLSM to dynamically decide when to fetch a new certificate, to distribute its load over time. Also, in case of scheduled sleep mode or connectivity loss, the NLSM can take this into account and act accordingly.

Any CSR received before the start of the renewal window is denied by the SLCA. This also makes sure, that certificate renewals are not spammed and that a node makes full use of the given certificate.

There is no set end time for the renewal window. *Node churn* may cause a node to be offline for some time, before being able to request a new certificate; so the node has ample time to rejoin the network.

Fixed Lifetime

The lifetime of every issued certificate is the same (for the same type of service). The maximum lifetime may be configured by the user over the UI. The mechanism can be combined with the renewal window policy.

Random Backoff

Whenever a certificate is generated, the SLCA sets the certificate lifetime to be lower than the maximum certificate lifetime, by a random amount. This is shown in Figure 4.4.

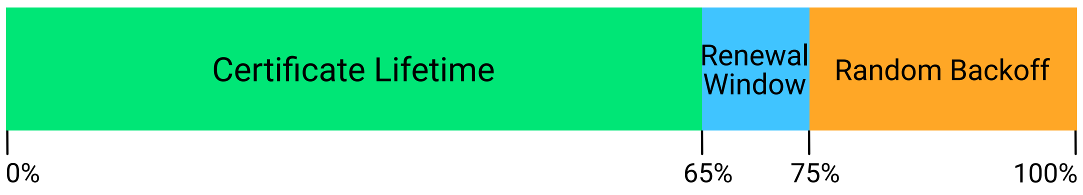


Figure 4.4: Random backoff mechanism combined with a renewal window

The random lifetime of the certificate is chosen in the timeframe:

$$lifetime_{max} * 0.75 \leq lifetime_{random} \leq lifetime_{max}$$

The introduced random backoff is, by default, up to a maximum of 25% of the maximum set lifetime. This parameter can be tweaked in the reference implementation.

The advantage of the solution is that every service, on every node, will require a new certificate at a random point in time. This allows for an improved distribution over time of the certificate renewals within a smart space. It partly addresses <R.16> but doesn't fulfill the requirement. The disadvantage of the random backoff mechanism is that the total amount of certificate renewals in the network increases proportionally to the cut-off time.
The implications of this mechanism are evaluated in chapter 6.

4.2.6 Certificate Revocation

Certificate revocation is necessary whenever access rights change, and in case a certificate is compromised. Revocation addresses <R.14> and <R.15>.

The system supports two different types of revocation mechanisms: an *active* and a *passive* revocation.

Active Revocation

The solution takes advantage of a publish/subscribe mechanism, offered by the VSL, to achieve active revocation. An NLSM subscribes to a VSL context node owned by the SLCA. The context node is specific for a service.

When certificate revocation is required, the SLCA publishes the information on the corresponding context node. This will notify the NLSMs subscribed to that info. Each notified NLSM will automatically invalidate the current certificate for that service and request a new one.

Active revocation has an immediate effect, but doesn't work for non-reachable (offline) nodes.

Passive Revocation

Passive revocation relies on short certificate lifetimes, introduced in subsection 4.2.4. It is meant to overcome the complexity of other solutions, such as CRLs and OCSP.

By leveraging short lifetimes, certificates are guaranteed to be revoked, at the latest, when they expire. This way, certificates on non-reachable nodes are automatically revoked when they expire <R.15>.

It is worth mentioning that a service without a valid certificate (or a revoked certificate) may still be run manually (i. e., without the aid of the SHE). However this service cannot communicate with any KA and cannot, therefore, exploit the functionalities of the VSL and the DS2OS in any way.

4.2.7 Clock Synchronization

For the automated certificate renewal and revocation mechanisms to work, clocks within the smart space need to be synchronized. This requirement is not hard, therefore there is no need to implement complex synchronization mechanisms.
A Network Time Protocol (NTP) [52] solution for a distributed system [85] is definitely overkill and requires unnecessary complexity.

The chosen approach to clock synchronization is very simple instead. The trusted anchor inside the site, i. e., the SLSM, dictates the time for all nodes.
Every NLSM periodically sends a **heartbeat** out-of-band message to the SLSM [58]. If the message contained a timestamp request, the SLSM responds with the current time, signed by the SLCA. When receiving the response, an NLSM notifies the SHE to update

the clock of the local node accordingly. It is important for the SHE to update the clock, as it is the entity providing the runtime environment and running every other μ-service.

The periodic heartbeat is used by the NLSM as a ping-pong mechanism and the newest timestamp is requested only once a day.
The clock synchronization mechanism does not focus on extreme precision, but allows to support the introduced features with negligible errors.

4.2.8 Key Management

Every node running an NLSM generates its own private/public key pair. The public key is published over the VSL by the NLSM and represents its ID within the network. This way, the SLSM can uniquely identify nodes and also authenticate CSRs.

Node Certificate Repository

Every node running an NLSM stores their certificates inside a dedicated `PKCS12` Key Store. The Key Store is password-protected.

The Key Store always contains:

- the root certificates of the SLCA and of the S2Store, needed to verify newly issued certificates;
- an arbitrary number of valid service certificates, corresponding to the number of services installed on the node.

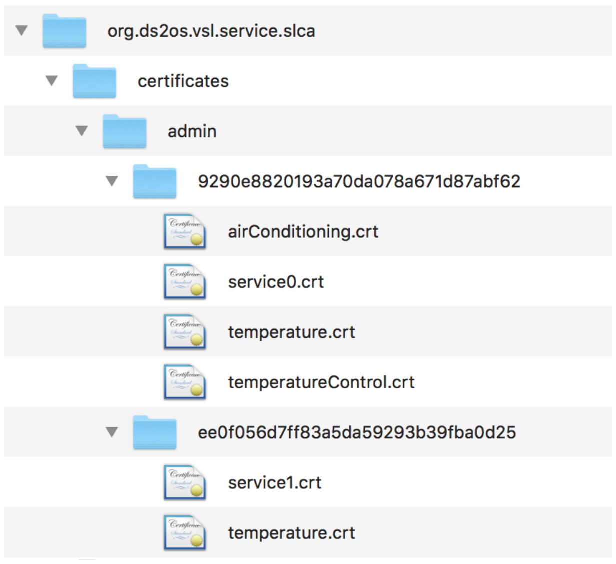
By design, all certificates are kept within the same Key Store, to avoid fragmentation and reduce management overhead.

The component in charge of adding/removing certificates from the Key Store is the NLSM. If, for some reason, the certificate of the NLSM or the SHE was not renewed in time, a new certificate must manually be issued and stored within the Key Store (see section 4.4).

SLCA Certificate Repository

The SLCA, on the other hand, must keep a copy of every valid issued certificate in local storage, for accountability reasons. Also, the current validity of the old certificate is checked whenever a new certificate is requested.

The certificates are kept within a certificate repository, accessible only to the SLCA. The folder structure is as shown in Figure 4.5. Multiple users are also supported, to allow different users to have different access rights and dedicated service instances.



```
▼ org.ds2os.vsl.service.slca
  ▼ certificates
    ▼ admin
      ▼ 9290e8820193a70da078a671d87abf62
          airConditioning.crt
          service0.crt
          temperature.crt
          temperatureControl.crt
      ▼ ee0f056d7ff83a5da59293b39fba0d25
          service1.crt
          temperature.crt
```

Figure 4.5: Folder hierarchy of the certificate repository managed by the SLCA

4.3 Service Management

Ultimately, the goal of this thesis is to protect the service management system from attacks. The service management is entirely based upon the components introduced in section 2.4.

The procedures that require dedicated protection mechanisms, presented in section 4.2, are the submission, download and installation of a μ-service. Service migration and uninstallation rely on the certificate revocation mechanism described in subsection 4.2.6, followed up by a new installation in case of service migration.

For the detailed service management logic refer to [12] and [58].

4.3.1 Interfaces

The relevant interfaces offered by the service management components are shown in the respective context models, in listings 5, 6 and 7.

The context nodes relevant for the protection mechanisms are:

```
<slsm type="/basic/composed" >
    <installService type="/basic/composed" reader="admin" writer="" />
    <uninstallService type="/basic/composed" reader="admin" writer="" />
    <certificate type="/basic/text" reader="nlsm" writer="" />
    <resources type="/basic/text" reader="nlsm" writer="nlsm" />
    <ipaddr type="/basic/text" reader="nlsm" writer="" />
</slsm>
```

Listing 5: Context model interface defined for the SLSM

```
<slca type="/basic/composed" reader="slsm" writer="">
    <certificate type="/basic/composed" reader="slsm" writer="" />
    <configuration type="/basic/composed" reader="slsm" writer="slsm" />
    <accessRights type="/basic/list" reader="nlsm,slsm" writer="slsm" >
            <user type="/basic/list" reader="nlsm,slsm" writer="slsm">
                    <service type="/basic/text" reader="nlsm,slsm",
                    ↪  writer="slsm" />
            </user>
    </accessRights>
    <addUser type="/basic/composed" reader="slsm" writer="" />
    <removeUser type="/basic/composed" reader="slsm" writer="" />
</slca>
```

Listing 6: Context model interface defined for the SLCA

```
<nlsm type="/basic/composed" >
    <startService type="/basic/text" reader="slsm" writer="" />
    <stopService type="/basic/text" reader="slsm" writer="" />
    <installService type="/basic/text" reader="slsm" writer="" />
    <uninstallService type="/basic/text" reader="slsm" writer="" />
    <stoppedServices type="/basic/text" reader="slsm" writer="" />
    <runningServices type="/basic/text" reader="slsm" writer="" />
    <resources type="/basic/text" reader="slsm,she" writer="" />
    <config type="/basic/list" reader="slsm,she" writer="" />
    <publicKey type="/basic/text" reader="slsm" writer="" />
    <ipaddr type="/basic/text" reader="slsm" writer="" />
</nlsm>
```

Listing 7: Context model interface defined for the NLSM

- `certificate`, taking in an encoded CSR. The same endpoint is offered by SLSM and SLCA, as described in subsection 4.2.3.
- `accessRights/<serviceId>` (offered by the SLCA), where NLSMs subscribe to, in order to be notified of certificate revocation.

4.3.2 Service Submission

Before a service can be downloaded and deployed to a smart space, it must be submitted to the S2Store.
Before submitting a μ-service, a developer must perform the following actions:

1. Upload their *public key* to the S2Store. This happens during a one-time registration process
2. Write a service manifest for the μ-service (see subsection 4.1.2)
3. Generate a developer-signed service certificate
4. Build a valid service package

Actions 2-4 can be performed using the `packager.jar` tool. The structure of a service package is described in detail in subsection 4.1.2.
A valid service package can be uploaded to the S2Store, where it will be validated, reviewed and eventually made public. This procedure corresponds to step 1 in Figure 4.1. The validation process includes adding a *store certificate* along the existing *developer certificate*, and repackaging the service. There is no automated mechanism for reviewing services yet, as the research project is still ongoing.

4.3.3 Service Download

Before deploying a μ-service within a smart space, it must be downloaded from the S2Store.
The implementation described in [12] considers an additional μ-service to be run on the orchestrator node: the *NLSM UI-server*. This service provides a web-UI accessible to the user for browsing μ-services available on the S2Store. Users can choose a service from the S2Store and download it to the local site. This corresponds to step 2 in Figure 4.1.

Once downloaded, the SLSM automatically takes care of verifying the integrity and authenticity of the service package. Invalid packages are discarded and the user is notified accordingly.

When a package has been verified, the user is asked to set up access rights for that specific service (see step 3 in Figure 4.1). These new access rights are then stored within the VSL.
If the downloaded service has dependencies, which weren't downloaded yet, these are downloaded as well using the same procedure.

4.3.4 Service Installation

Downloaded services with configured access rights can be deployed on nodes within the smart space. The SLSM does this automatically, without any user intervention. Specific load-balancing algorithms and policies are used for this purpose. For example, a service may have specific hardware requirements and not be able to run on any node.

Once a target node has been identified, the following operations occur:

- The SLSM initiates a GET request on the `installService` endpoint of the NLSM running on the target node
- The NLSM opens a dedicated transmission control protocol (TCP) socket on a random port and tells the SLSM the number of the port
- The SLSM then connects to the socket and sends the service package, retrieved from the local service repository (see step 4 in Figure 4.1)
- The NLSM on the target node verifies the integrity and authenticity of the service
- The NLSM requests a *site certificate* for the new service (see step 5 in Figure 4.1).

Once the service has a valid certificate, the NLSM instructs the SHE to start it.

4.4 System Bootstrap

The original DS2OS design assumes that edge nodes deployed within smart spaces are already pre-configured and have all the necessary software running on them.

This assumption may be valid in an industrial approach, where IoT devices are setup with default configurations. For private consumers, this is unlikely, especially considering the heterogeneity of existing devices. Edge nodes may come without any software pre-installed at all, or just with a pre-installed OS. Most importantly, services running on nodes require certificates issued by the SLCA, or they won't run at all.

From a user perspective, it is unpractical to set up an edge node manually. The procedure would require an expertise that most users do not possess.

This section introduces a simple system setup concept, valid for any new node within the DS2OS. <R.18> is addressed in this way. The only requirements for such a solution to be feasible and automated are:

- the device must run a Unix-based OS with a Secure Shell (SSH) server enabled (or equivalent channel)

- the user must know the credentials of the device
- the device must have Java installed. This could also be integrated in the automated process, but is less relevant for a general-purpose solution

4.4.1 Setup Orchestrator Node

The orchestrator node is considered to be a trusted node. When setting it up, it is assumed that the node is not compromised in any way.

The setup process requires downloading a *starter pack* from the S2Store, containing the following bundles: *KA*, *SHE*, *SLSM*, *SLSM UI-server* (see subsection 4.3.3), *SLCA* and *NLSM*. Additionally, a *bootstrap* program to startup the system is also included. All these bundles and applications must be signed and verifiable. The KA is the biggest source of worries, so it requires an S2Store-issued certificate as well.

Once downloaded, the execution of the *bootstrap* program performs the following operations:

1. All downloaded software bundles are checked for integrity and authenticity
2. The user is prompted for a new administrator password, which will be valid for managing the smart space
3. The local network settings are checked, and a `config.txt` file for the KA is generated
4. A new CA key is generated and an intermediate certificate is requested by contacting the S2Store
5. The directory structure for the downloaded services is created
6. Using the `openssl` tool (or equivalent), a short-lived certificate is generated for the downloaded services. A new certificate will be issued by the SLCA as soon as the NLSM runs, replacing the current one
7. The SHE is started, which will automatically run the KA, the NLSM, the SLCA, the SLSM and the SLSM UI-server (as described in [58]).

Once the UI-server service is running, the user can interact with the system using a friendly UI. The current implementation of this additional service is described in [12] and offers a web-based UI.

4.4.2 Setup Generic Node

Any new node joining the network can be setup and configured using the SLSM UI-server. The setup process is automated, as visible in Figure 4.6, and the user only requires to start it via the UI. A discovery mechanism would be needed in the future, in order to find new IoT devices that require to be set up.

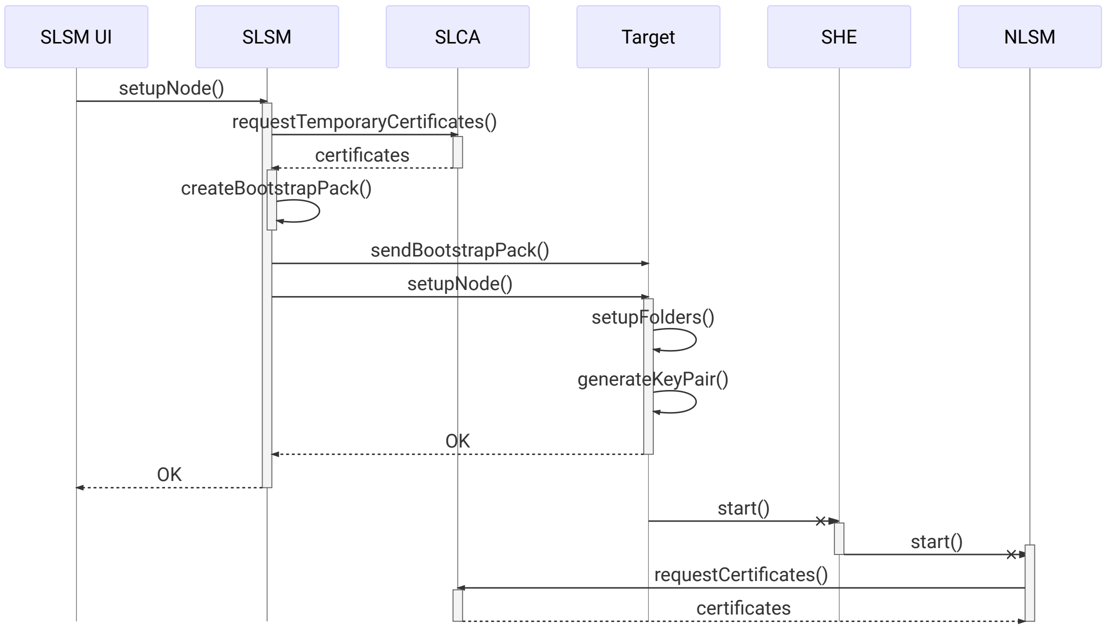


Figure 4.6: Setup of a remote node using an automated bootstrap mechanism

The bootstrap package sent to the target node contains everything the node requires to start the VSL and the services, including temporary certificates for the SHE and NLSM.

The setup procedure takes place on the node and involves checking network settings, generating configuration files and so on. The SHE and NLSM shown on the right are the instances started asynchronously on the target node.

Key stores on every node are password-protected. Since nodes are independent and unattended, the password needs to be stored locally in a configuration file, on each of them. For this reason, it is strongly advised to use a different password for each node. This way, if a single node is compromised, the are nodes remain unscathed.

Chapter 5

Implementation

The chapter provides details about the prototype implementation of the solution designed in chapter 4.

Section 5.1 briefly describes the implementation process and the structure of the code. Section 5.2 focuses on the workarounds and limitations of the current implementation. Finally the configuration parameters of different entities are covered in section 5.3 and section 5.4.

5.1 Code Structure

The designed solution was implemented using Java in collaboration with other two students. The themes had some overlaps and required appropriate organization and task management.
The responsibilities during this phase were shared equally among all participants.
The code is available at `https://gitlab.dev.ds2os.org/ds2os-devs/service-management`.

The entire solution was split in different Java modules: one module for each management μ-service, plus a common module, containing shared logic, utilities and interfaces.
The common module contains most of the security algorithms and primitives, required throughout the system.

The API used for all security operations is contained in the *Bouncy Castle Crypto* library [37]. This library is the main Java reference library for all cryptographic operations. Furthermore, it is the only library natively supporting X.509v3 extensions.
For managing *Key Stores*, the built-in Java API is used.

The common module is used as a direct dependency by the SLSM, SLCA, SHE and NLSM modules. This approach enforces maximum code reusability.

Each μ-service implements the logic described in [12] and [58]. The security logic is implemented on top of the basic DS2OS management functionalities.
Leveraging the defined interfaces (see subsection 4.3.1), each security-oriented block can easily be replaced by a different concrete class in the future.

The designed bootstrap system, introduced in section 4.4, was not implemented along the rest of the solution. It is, therefore, to be considered as future work.

5.2 Issues & Known Workarounds

Several workarounds were necessary, in order to get the proposed solution to work with the current implementation of the VSL.

Base64

`GET/SET` requests on the VSL don't support raw binary data. For this reason, CSR requests and certificates are encoded in *base64* format. This workaround was necessary, to avoid using side-channels for the security mechanisms. Every side-channel opens up an additional attack vector, which is avoided in this work.

The downside to this approach is an increased payload size during transmission.

Key Stores

The VSL connector requires service certificates to be passed in a standalone Java *Key Store*.
The Key Store can only contain one service certificate and one root certificate, as no alias comparison is used by the connector. As a result each service certificate must be stored in a separate Key Store. Also the root certificate is contained in each Key Store. Every node and μ-service communicating with the VSL is affected by this. The side effect is an increase in used storage space and I/O operations, as well as higher management complexity.

In a future version of the DS2OS (and VSL), it should be possible for each node to store all certificates within the same Key Store.

OID Restriction

The current VSL implementation already uses a custom temporary OID for checking the access rights of a μ-service. The OID is `1.3.6.1.4.1.X`, with `X` being an incremental suffix. This OID is officially reserved, and should therefore not be used.
The OIDs presented in subsection 4.2.2 can be considered future work, while the current implementation adapts to the OIDs used by the VSL.

5.3 KA Configuration

KAs require a `config.txt` file, to read node-specific configurations. Configuration parameters are specified in `<key=value>` pairs. An example configuration is visible in listing 8.

```
kor.archive.enabled=0
kor.db.persist=0
kor.db.username=ds2os
kor.db.password=thisMustBeSecure
kor.db.location=hsqldb/db
cache.enabled=0
transport.interfaces=en0
alivePing.senderInterval=3
transport.allowLoopback=0
modelRepository.localPath=models
```

Listing 8: Example configuration required by a KA

The `kor.db.persist` line must have value 1 for the node hosting SLSM and SLCA. This allows to persistently store the access rights and other settings configured by the user.

The configuration file is automatically generated by the SHE during the first deployment process (see section 4.4). This is essential to set the correct transport interface to be used in the future.

5.4 μ-service Configuration

Every service takes a dedicated configuration file as input. Configuration files are stored in the same format as KA configuration files (see section 5.3).

Every configuration file is expected to contain the mandatory fields shown in listing 9.

```
serviceKeystore=ssl-keys/slsm.jks
workingDirectory=run/org.ds2os.vsl.service.slsm
agentUrl=https://192.168.0.142:8081
```

Listing 9: Example minimal configuration required by a generic μ-service

The `workingDirectory` represents a path in the file system, where the service has write access and can store data.

The `serviceKeystore` is the Unified Resource Identifier (URI) of the service certificate.

The configuration files on distributed nodes are automatically generated by the respective SHE when the target service is first installed and set as read-only files.

Additional configuration parameters can be set over the VSL. This is done on a per-service basis, by the SLSM.

SLCA Configuration

Additional configuration parameters required by the SLCA are:

- `caCertificate`, containing the path of the root certificate
- `caKey`, containing the path of the private key tied to the root certificate, used for signing all service certificates
- `certificateValidity` contains the maximum validity of a certificate in seconds (e. g. 7200 for 2 hours validity)
- `renewalPolicy` can be either *random* or *default*. If random, then the `randomBackoff` parameter is required
- `randomBackoff` is an integer in the range 0-100, containing the coefficient to be applied when using the random backoff policy

SLSM Configuration

Additional configuration parameters required by the SLSM are:

- `networkInterface`, which must be the same one used by the KA. It allows the SLSM to publish its own Internet Protocol (IP) address, which the NLSM use for sending heartbeats.

Chapter 6

Evaluation

The chapter contains an evaluation of the designed solution, along with important considerations regarding the impact on IoT nodes.
The evaluations is two-fold:

1. the security against attacks is assessed in, indicating which implemented countermeasures protect against which kind of attack
2. the performance of the solution is evaluated in section 6.2, with a focus on additional resource consumption and scalability.

6.1 Security

The designed solution didn't undergo a formal security review, but considering the widespread use of certificates, the security against tampering and the authentication guarantees can be assumed. The main attacks on the IoT were originally analyzed in section 2.5.
Table 6.1 summarizes these attacks once again. For each attack, the table shows whether and how the designed solution offers protection against it.

As expected, the usage of PKI and certificates protects against application-layer attacks, such as malicious code and malwares. The introduced integrity and authenticity checks guarantee, that only trusted apps are downloaded from the S2Store and run.
Unauthorized conversation and access to resources is neutralized thanks to the usage of PKI certificates, with embedded access rights.
The proposed bootstrap mechanism allows only the administrator to introduce new

nodes in the network, therefore offering protection against node impersonation and sybil attacks.

Attack	**Layer**	**Security Measure**
Hardware Trojan	Phy	None
Node tampering	Phy	None
Node impersonation	Phy	Bootstrap mechanism and usage of PKI authentication forbids node impersonation. Keys can be stolen though, if the attacker has physical access to a node
Eavesdropping	Net	Encryption provided by the VSL
MiTM	Net	Secure channels provided by the VSL
Routing	Net	Encryption provided by the VSL, but no advanced routing techniques are in place
Sybil	Net	Authentication, centralized control on SLSM and bootstrap mechanism deny the forgery of new identities
Unauthorized Conversation	Net	Every entity is authenticated within the DS2OS; controlled access is enforced via certificates
Virus, Malware and Spyware	App	Only verified μ-services can be installed on nodes. Every μ-service is code-signed. Authenticity and integrity are checked when a service is started
Malicious code injection	App	Applications are enforced to use controlled DS2OS interfaces
Protocol	App	Applications are enforced to use controlled DS2OS interfaces. No further protection for app-specific protocols is in place
DoS	Any	Application-level authentication. No protection on network layer
Social Engineering	Any	None
Side-channel	Any	None

Table 6.1: Attacks on the IoT physical layer and countermeasures

6.2 Performance

The design of the solution introduced in chapter 4 revolves around digital certificates. Certificates are more heavyweight than tokens, and using them as main protection mechanism has its tolls on the performance of the system [69]. Furthermore, performance and scalability are major concerns when using short lifetime certificates (see section 3.3) and automated certificate renewal.

Another relevant metric for evaluating this solution is the analysis of the energy consumption and computational load on IoT devices. These resource-constrained devices are the target nodes of the solution.

With the proposed solution, the automated certificate renewal is the only security mechanism that needs to be evaluated. No further mechanisms are in place, making the solution simple and consistent.

The evaluation focuses on the effects of the proposed automated short lifetime certificate-based solution in different ways:

1. The effect of different certificate lifetimes on the overall network traffic is evaluated
2. The positive effects of introducing a random backoff to the renewal intervals is analyzed
3. The computational load and the energy consumed is measured in a representative IoT setting
4. Finally, the scalability of the solution is evaluated, with respect to SLSM and SLCA. These represent the bottlenecks of the solution, as they are unique within the network. For this purpose, the throughput under load of the SLCA is analyzed.

6.2.1 Setting

Software Components

For the evaluation, the reference implementation of the VSL and the implementation described in chapter 5 were used.

All running software, besides the KAs, is implemented as a μ-service. All μ-services communicate with each other over the VSL REST interface using HTTPs.

To emulate a real smart space, the deployment shown in Figure 6.1 is considered. Each node within a local network runs a KA, an SHE, an NLSM and multiple μ-services.

One coordinator node within the network additionally runs the SLCA and SLSM. Combining the two entities that frequently communicate increases the performance.

Hardware Testbed

All tests were performed using real devices, software and processes. Two different testbeds were used.

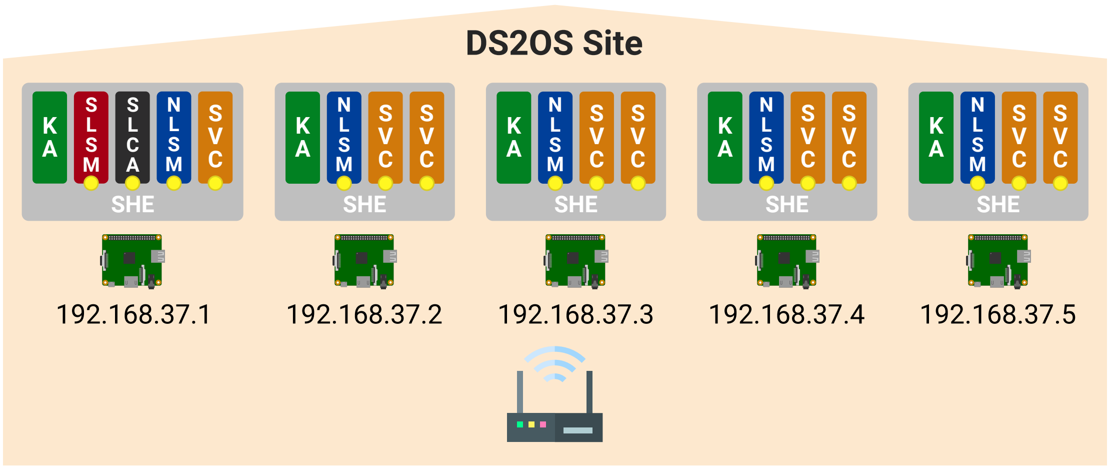


Figure 6.1: Emulation setting with 5 nodes, one of which is the orchestrator node (left-most node)

For the **network performance** tests (see subsection 6.2.2), a setting of 5 desktop-class computers was used, with the specifications shown in Table 6.2 on the left. This setup is most likely beyond the resources of IoT installations in the next years. However, it enables to evaluate the effects of the solution better as performance side-effects would not play a big role. For actual IoT hardware those could heavily bias the measurements.

For the **computational and energy consumption tests**, it made sense to use IoT hardware. The tests were performed using two *Rasberry Pi 3 Model B* as representative IoT devices, with the specifications shown on the right of Table 6.2. In addition, several desktop-class computers (specifications shown in Table 6.2) were used for generating more load towards the IoT nodes. This had the purpose of stressing out the IoT nodes to the maximum.

CPU	Intel Xeon E3-1265L V2 @2.50 GHz
RAM	16GB
Storage	SSD
OS	Debian 9

CPU	ARM Cortex A53 @1.20 GHz
RAM	1GB
Storage	Samsung micro-SD
OS	Raspbian Stretch

Table 6.2: Testbed hardware specifications; left: computer used for emulation; right: Raspberry Pi 3 used as IoT node

Network Configuration

All tests were performed in a LAN network with a 300Mbps router. The network bandwidth and speed was not limited, in order to have results close to those observable in a typical LAN network, regardless of the used devices.

Traffic generated by the VSL is encrypted over HTTP and therefore very hard to analyze. For this reasons, the data was collected logging the size of a request/response to a file. Since the communication between the nodes happens over HTTPs, the measured data doesn't take the protocol overhead into account. The header of every HTTP request/response, however, will always be the same and can therefore be ignored.

Communication between the SLSM and SLCA happens locally on the same host. The overhead of this additional operation is considered negligible for the purpose of the network evaluation, as local communication doesn't have latency limitations.

Tested μ-services

Only dummy services were used for the purpose of this evaluation. The services do not perform any active operations and stay idle forever. This allows to evaluate the raw performance impact of the automated certificate renewal, without external influences or spurious data. The amount of services used for each test is variable, depending on the test itself.

When an emulation is started, all services are already installed on every node within the network, but have no valid certificates. All services on each node are always started sequentially, as soon as the NLSM is running.

Certificate Size

Service certificate sizes in the DS2OS depend on various factors:

- the key length
- the name of the service
- the access IDs

The assumed key length is 2048bit. The names of the services used for the tests are: *temperature*, *airConditioning*, *temperatureControl* and other dummy services named *service0* through *serviceN* (N depending on the amount of services). The access ID set for all services is the name of the service itself, with the exception of the *temperatureControl* service, which also adds the *home* access ID.
Given these considerations, the certificate sizes of the services turn out to be mostly identical, with some minor differences of only a few bytes.

6.2.2 Network Impact

The network performance evaluation considered different real-time emulations on 5 nodes, each running 30 μ-services, for a total of 150 μ-services running in the site.

Optimal Certificate Lifetime

The first evaluation aimed to identify a suitable certificate lifetime for further testing and normal usage. For this purpose, the average traffic consumption with different certificate lifetimes was measured: 5, 10, 20, 30, 60, 90 and 120 minutes. The same test was performed for both validity policies: fixed lifetime and random backoff. The maximum random backoff was 25%. The chosen renewal window in both cases was of 30 seconds.
Each emulated setup was run for 5 certificate renewal periods, without any time acceleration.

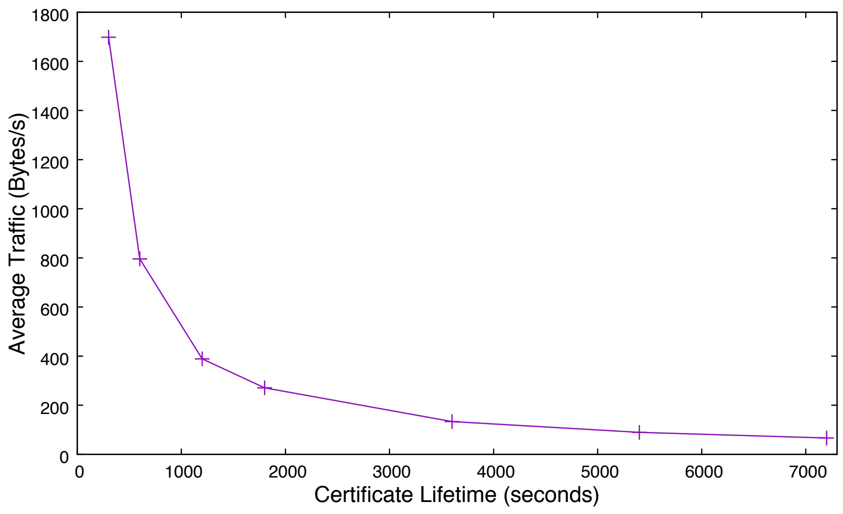


Figure 6.2: Total average traffic consumption as the certificate lifetime increases

As expected, the average traffic decreases inverse proportional the longer the certificate lifetime is. Figure 6.2 shows this clearly.
For prolonged period of times, the average traffic would increase by up to 12% for the solution with random backoff enabled. This small increase is negligible, considering the already low amount of generated traffic.

From this measurement, 3600s was identified as good compromise between short lifetime and low traffic. A higher validity can be chosen in IoT scenarios, where the bandwidth limitation is more severe. This is not the case for the envisioned scenario, therefore 3600s (1 hour) certificate lifetime was used for following tests.

Traffic and Random Backoff

The main measured metric, for analyzing the network impact, was the instantaneous network traffic. This was done from the point of view of the orchestrator node, running the SLSM and SLCA.

Figures 6.3 and 6.4 show a scatter plot of the total traffic produced by certificate renewals at any point in time. In both plots all services are started roughly at the same time. Starting all services at the same time is realistic, e. g. after a power outage.

For the case with fixed certificate lifetimes, this results in periodic traffic bursts, as visible in Figure 6.3. These traffic bursts cause an instantaneous traffic of up to 50KB per second. This is realistic, considering the size of CSR is 810 Bytes on average, while the size of a certificate is 1600 Bytes. This amount of data is not entirely optimized, due to the usage of *base64* encoding (see section 5.2). The maximum amount of instantaneous traffic generated is capped by the SLCA, which can only process a limited amount of requests per second.

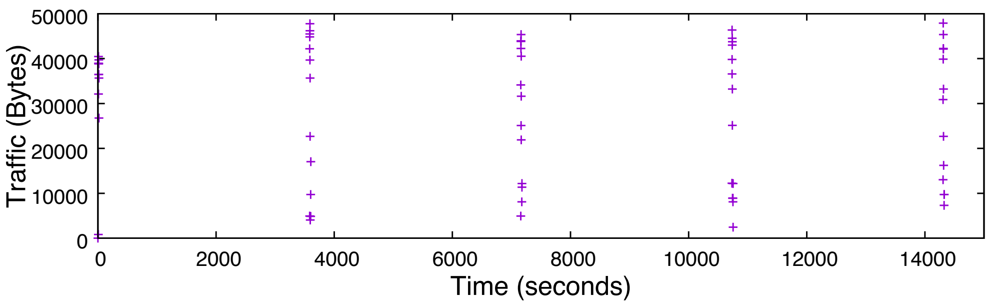


Figure 6.3: Traffic generated by 150 μ-services using fixed certificate renewal time

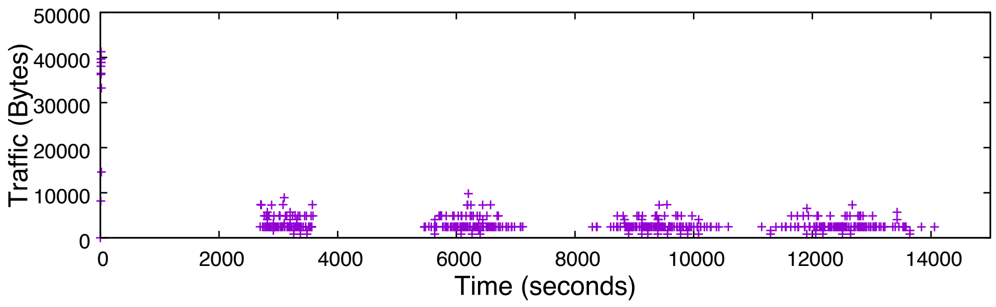


Figure 6.4: Traffic generated by 150 μ-services using certificate renewal with random backoff

Introducing the random backoff allows to mitigate the traffic from unwanted spikes. The backoff is chosen by the SLCA after each renewal. Setting a random backoff of 25% turned out to be a good trade-off between load distribution and minimizing unwanted additional traffic that is caused by the resulting additional certificate renewals.

Figure 6.4 proves that, as soon as the random backoff mechanism kicks in, the traffic spikes disappear. After about 12000 seconds, the traffic is well distributed resulting in peaks of at most 10KB. This behavior is desired for the IoT where instantaneous bandwidth is limited (e. g. 2G/3G networks).

To verify the distribution, the overall traffic average over both emulations was calculated, after a runtime of 20 hours. It was 119 B/s for the simulation driven by a random-factor and 106 B/s for the other case. The slight delta between the two is introduced by the random factor, which causes more certificate renewals within the experiment run time, as services do not wait until a certificate is almost expired for requesting a new one.

Node Churn

Node churn was emulated by randomly disconnecting a node from the network.
The resulting impact of node churn on the network depends on the frequency and length of the downtime period. Considering a $length > certificate_{lifetime}$, node churn will always cause a sudden spike in traffic, due to the node requiring new certificates for all services after re-joining.
For the solution with fixed lifetimes, this actually distributed the spikes over time, introducing a positive effect. For the solution with random backoff, a prolonged node downtime would mean a sudden isolated spike in the traffic, introducing an undesired effect. This effect would be mitigated by the random backoff immediately afterwards.
If a node is absent from the network for a shorter period of time, the consequences are typically very limited.

6.2.3 Impact on IoT devices

The IoT is resource limited. Key resources are compute power, network bandwidth, and energy consumption. To measure them, a *Raspberry Pi 3 Model B* device was used.

Network bandwidth usage was already evaluated in subsection 6.2.2. Two main metrics were analyzed on IoT devices: computational load and energy consumption.

Two tests were run for this purpose:

1. The Raspberry Pi hosted an NLSM instance and run 20 services

2. The Raspberry Pi acted as the orchestrator, hosting only the SLCA and SLSM (no further services). 3 other nodes, running 20 services each, were deployed on desktop computers

For both tests, very short certificate lifetimes were used (5 minutes maximum validity period). Random backoff was also enabled. This allows to see the distribution of the power and CPU spikes, caused by the random backoff mechanism.

The results in Figure 6.5 and 6.6 show the energy consumption on the left y axis, and the CPU load on the right y axis. Spikes are clearly visible in the CPU load and the energy consumption where the certificates get replaced and verified.

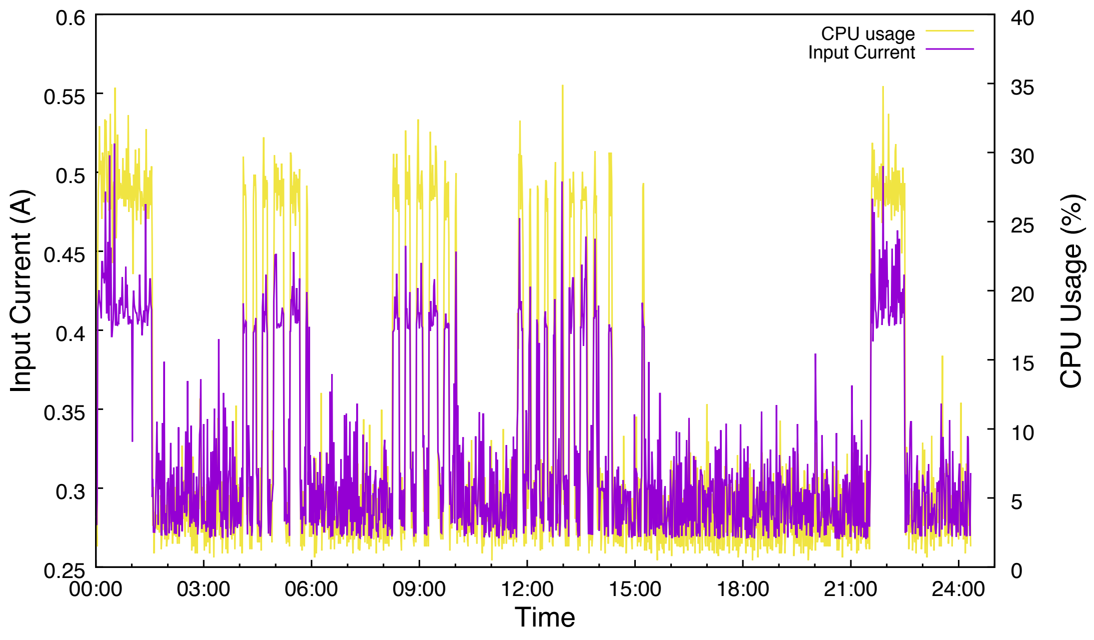


Figure 6.5: CPU load and energy consumption for an NLSM deployed on an a Raspberry Pi

Computational Load

The CPU load was measured using a custom python script and the `psutil` library. The script continuously monitors the CPU usage in background and logs the current average load with a 1 second interval. The values include the sum of the load on 4 CPU cores.

The measurement in Figure 6.5 shows that the total CPU load during certificate renewals reaches about 30%, saturating one core on the Raspberry Pi. On the node hosting the SLCA (see Figure 6.6), the load tends to be higher (up to 40%). This is due to the

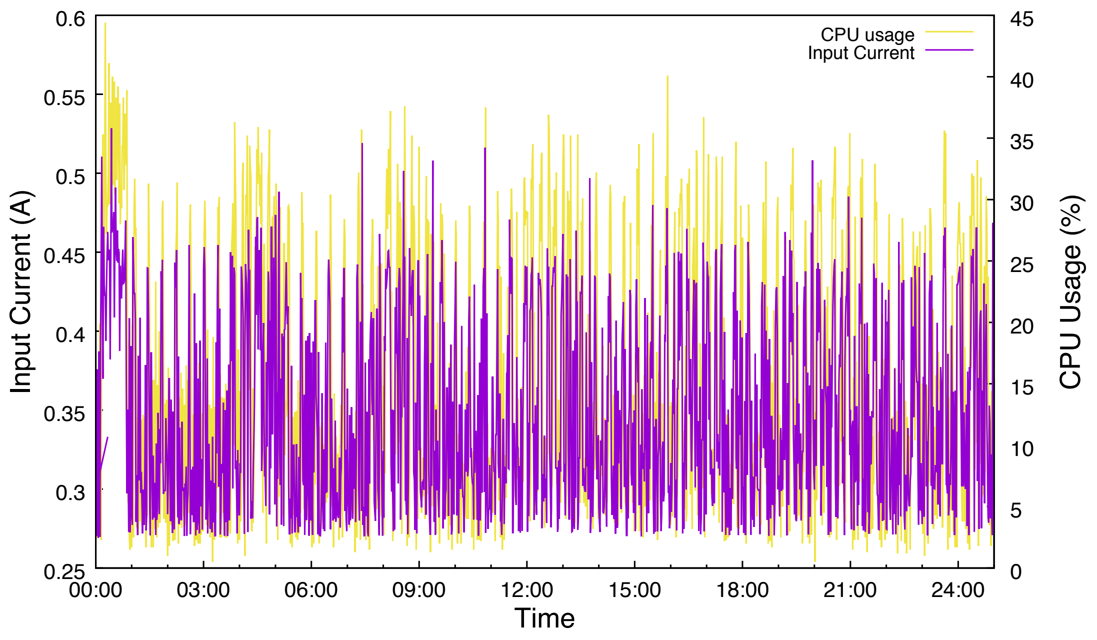


Figure 6.6: CPU load and energy consumption for SLSM and SLCA deployed on an a Raspberry Pi

SLSM also being deployed on the same node and performing management operations in the background.

The SLCA, is visibly overloaded during the initial phase, when all nodes require new certificates. As soon as the random backoff kicks in, the CPU spikes start distributing over time.

Apart from the CPU load, an important finding was the Input/Output (I/O) limitation introduced on the Raspberry Pi. Because of the low speed in storing a new certificate inside a Key Store and verifying its metadata, a full certificate renewal takes about 2.5s on the Raspberry Pi. This is significantly slower than on a desktop-class node.

In case of fixed certificate lifetimes, this could lead to prolonged service downtimes, in case a node is running many services with expired certificates. With random backoff enabled, however, the renewals are more distributed over time and are less likely to represent a bottleneck.

After roughly 15 minutes in Figure 6.5, a **node churn** was simulated, disconnecting the node from the network. Upon node churn, the device cannot request new certificates. As soon as the device is reconnected to the network (after minute 21), all certificates have expired and need to be renewed. The sudden increase in traffic and power consumption can clearly be seen in the plot.

Similarly to the reasoning made for the bandwidth consumption, the random backoff mechanism introduces up to 25% more certificate renewals. Consequently, this also leads in a slight power consumption increase over time.

Energy Consumption

The energy consumption was measured using an UM25C voltmeter/ammeter [83]. The UM25C was set between the power outlet and the Raspberry Pi. The UM25C continuously measured the current voltage and drawn current of the Raspberry Pi. The samples were taken by the UM25C in a 1 second interval, measuring the average value within the time period.

When idle, the Raspberry Pi device draws approximately 250mA. This is clearly visible in Figure 6.5. Heartbeats and other VSL low-level periodic mechanisms lead to an increase in power consumption by 50 to 100mA.

The main energy consumption spikes, however, occurs during a certificate renewal. This is clearly visible for both the NLSM and SLCA cases. Cryptographic operations (CSR and certificate creation) are currently not hardware-accelerated and are executed entirely in software. The node requires significantly more power in this state, between 400 and 500mA.

When multiple certificates need to be renewed at the same time, an NLSM requests them sequentially, therefore not exceeding a certain power consumption threshold. The same reasoning applies to the SLCA, which processes certificate requests sequentially.

6.2.4 Scalability

Scalability is analyzed only for IoT devices, as they represent the target of this thesis.

The presented solution targets IoT nodes capable of running Java, but still with constrained resources. Target devices are single-board computers capable of running Linux, such as Raspberry Pi, Beaglebone, or Intel Galileo. To prove that the solution runs without excessively increasing the load on existing devices, the following was analyzed:

1. the time taken by an NLSM to renew a certificate on average
2. the amount of certificate renewals that the SLCA could handle per second

For the purpose of this evaluation, automated certificate renewals with random backoff mechanism were considered.

NLSM

The main scalability issues introduced for the NLSM are given by the slower I/O and cryptographic operations. Given the results, some limitations need to be considered.

T represents the time (in seconds) needed by the NLSM to renew a single certificate. For certificate lifetime M seconds, the total amount of feasible renewals over a period M is:

$$M/T$$

For a Raspberry Pi with $T = 3$ and $M = 3600$, the maximum amount of supported services before slowing down is 1200. Furthermore, the service availability isn't affected only if the renewals are perfectly distributed over time.

These considerations are theoretical and consider an ideal system. While an NLSM is renewing a certificate, it is busy and cannot perform any other active operation. In a real scenario, the NLSM is in charge of other tasks and cannot be busy 100% of the time renewing certificates.

For the above parameters, running only 10 services would cause the NLSM to be busy renewing certificates about 1% of the total uptime.

The best optimization strategies to consider on an NLSM are:

- The total amount of services running on the node must be strictly $\leq M/T$
- Installed services should not require a new certificate all at once
- Scale the amount of services running on the node according to the imposed M and T of the device

SLCA

The node running the SLSM/SLCA represents a potential bottleneck, since it has to manage all other nodes.

The results reveal that, on a desktop-class SLCA node, the amount of certificates that can be renewed per second is roughly 15. On a Raspberry Pi, however, this number goes down to 2, with an average processing time of 500 milliseconds per request. This data imposes some hard limits on the setup in an IoT setting.

N represents the maximum amount of certificate renewals that the SLCA is capable of processing per second, before throttling. N increases as the resources of the SLCA increase. For $certificate_{lifetime} = Nseconds$, the total amount of feasible renewals over a period M is:

$$N * M$$

This is only true, if the renewals are uniformly and ideally distributed over time. If every node runs the same amount of services, the total amount of supported services by each node is:

$$(N * M)/K$$

In the tested scenario, with $M = 3600$ and $N = 2$ on the Raspberry Pi, $K = 100$ nodes could potentially run up to 72 services each.

These considerations are theoretical and consider an ideal system. In reality, other running services take up bandwidth and the node running the SLCA and SLSM might be busy performing other tasks, such as installing a new service or migrating one. Also, this ideal scenario would saturate the load on the orchestrator node all the time.

In order to keep the system reactive and avoid queueing certificate renewal requests, the best strategies are:

- The total amount of services running in the system must be strictly $\leq N * M$
- The random backoff policy helps distributing the renewals over time. As the requests become more distributed over time, the necessity to have a higher request throughput on the SLCA diminishes
- The more services need to be deployed in the system, the longer the certificate lifetimes should be, to improve the overall performance

6.2.5 Comparison with Authentication Tokens

The metric that needs to be tested when comparing the automated certificate renewal with authentication tokens is the communication overhead introduced by the two different approaches. This comparison is purely mathematical and the same setup for the two solutions is considered. A small-sized JWT token with only a couple access IDs as payload is assumed, with a token length of around 80 bytes. Refresh tokens needed for renewing access tokens can be smaller. 40 bytes will be considered in this case.

Previous tests showed that a single certificate renewal produced 2.5 KB of traffic. Compared to a total of 120 bytes for a token refresh, the difference is huge.
However, access tokens are transmitted over HTTPs in every request, when nodes communicate. This overhead is added on top of the underlying TLS connection. This means

that if a μ-service were to perform at least 20 requests to other services within the smart space, during the period of validity of a token, the two solutions would be balanced out in terms of total communication overhead. The more communication between nodes is expected, the better the certificate-based solution performs against a token-based approach. To perform a more realistic comparison, additional information about the smart space usage is required.

6.2.6 Comparison with OCSP and CRL

The automated certificate renewal is introduced as an alternative to other certificate revocation schemes, such as OCSP and CRL. The purpose was keep the revocation mechanism as simple as possible, without introducing management overhead. The downside is an increased communication overhead.
A mathematical comparison between the communication overhead over time, produced by the existing revocation schemes, is now presented. The work presented in [3] is taken as reference.

In CRLs, a request size of roughly 100 bytes is considered, while a response contains the list of all revoked certificates, signed by the CA. A signature is around 700 bytes long, while each revocation entry is as long as the serial number of the revoked certificate. A serial number is typically 20 bytes long.
In OCSP, the average request is considered to be 140 bytes long, while the average response is 150 bytes long. For each certificate in the system, an OCSP request needs to be sent to the server.

The comparison assumes a setting with 5 regular nodes running 10 services each. To make a fair comparison, the same revocation check frequency of 1 day is assumed: CRLs are fetched once per day by every node; OCSP requests are sent for every running service once per day; for the designed solution, certificates are renewed once per day.
If no certificates are revoked during the day, fetching CRLs generates 4 KB, and updates via OCSP generate 14.5 KB of traffic. Certificate renewals, on the other hand, generate a total of 125 KB of network traffic per renewal.

In such a scenario OCSP scales very well, while the certificate expiration approach outperforms CRLs only after roughly 6000 certificates have been revoked. This is hardly to be expected in smaller smart spaces. However, if the considerations of subsection 6.2.5 are take into account, the certificate expiration approach becomes very valid: the scalability is comparable, but the management complexity is minimized.

Chapter 7

Conclusion

The goals of the thesis were to analyze and improve the security of an IoT management system, such as the DS2OS.
The main focus was the protection of applications distributed within a smart space.
This included guaranteeing the integrity, authenticity and controlled access of the services, along with securing their lifecycle management. This conclusive chapter quickly covers the results and future works.

7.1 Results

Based on the analysis of the scenario and the existing threats in the IoT world, a PKI-based approach was taken. While this approach seems less suitable for constrained IoT devices, the evolutionary pace of these devices points to hardware capable of supporting such features in the very near-future.
The proposed solution combines a periodic authorization refresh concept, used in token-based systems, together with a code signing mechanism.
Applications in smart spaces are protected by pinning a certificate to them. The certificate is signed by a trusted entity in the smart space.
To enforce access control policies and mitigate certificate compromise, an automated certificate renewal process was introduced.

The designed solution integrates seamlessly with the DS2OS. Certificates go hand-in-hand with TLS communication channels, provided by the VSL. At the same time, the same certificates are used for fulfilling the required security goals.
The application lifecycle management system implemented in [12] [58] is automatically

protected by the proposed security features. The system allows only trustworthy services to be run, similarly to smartphone ecosystems, but in a distributed scenario.
The implemented solution proved to be effective, with a very low design complexity.

The results obtained from the evaluation show that the overhead network traffic generated by the solution is mainly attributed to the periodic certificate renewal.
This overhead proved to be, in fact, quite limited and easy to mitigate by choosing longer certificate lifetimes. The trade-off between increased security and lower overhead represents an important factor to be considered in a real smart space.
The main bottleneck of the solution was attributed to the CA running on a device with limited hardware resources. This will be addressed in future works.

7.2 Future Work

Some important aspects, that may be addressed in the near future, are:

- The implementation of the system bootstrap procedure described in section 4.4
- Adapting the VSL implementation to better support the designed security features
- A UI for managing the user access rights needs to be implemented

Optimization of the solution is a broader topic and requires further investigation.
One possible solution could be the usage of implicit certificates (see performance impact in subsection 3.3.1).
A more radical approach to overcome issues identified in the evaluation would be to address the bottleneck introduced by the single CA. Leveraging multiple CAs could lead to a fully distributed system. Such a scenario introduces further research questions, such as the trustworthiness of additional CAs and how to manage network partitioning. While the complexity of the system would increase, so would the scalability.

Another possible research topic is the automated handling of invalid CSR or certificates. Using machine learning and other techniques it could be possible to identify threats and handle them accordingly, with or without user intervention.

Chapter A

Appendix

A.1 Site Local Service Certificate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1533934650097 (0x16525a142f1)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=DE, ST=Germany, L=Munich, O=TUM, OU=I8, CN=DS2OS CA
        Validity
            Not Before: Aug 10 20:57:30 2018 GMT
            Not After : Aug 10 21:07:30 2018 GMT
        Subject: C=DE, ST=Germany, L=Munich, O=TUM, OU=I8, CN=temperature
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b3:2c:5f:bb:cf:7f:98:c3:ea:19:02:0f:ec:80:
                    d7:0e:4b:09:35:3e:39:7f:79:50:d6:42:0d:82:ff:
                    32:40:28:45:f8:c6:cf:7d:2e:c6:e7:1c:4a:ca:50:
                    47:36:d7:1f:25:b5:bd:f7:c2:0b:86:03:69:f3:a6:
                    09:95:f7:25:73:1c:68:bd:1e:ce:ba:26:4e:f0:d4:
                    e6:32:12:62:dc:2a:f7:15:2e:cc:97:0c:6e:43:73:
                    15:4f:6f:04:43:65:35:2a:91:14:6e:b2:c0:ca:f3:
                    1a:66:d9:f8:4c:a6:e8:f0:4d:5b:cc:12:6c:11:c3:
                    04:a7:35:e7:0b:ab:db:dd:90:41:86:72:d3:b9:bf:
                    4f:04:33:f1:c6:dc:2f:fc:b6:bc:07:48:c3:76:34:
                    b1:5d:9c:b1:cf:9a:5b:e9:ae:f5:85:d5:5f:86:cc:
                    d6:1d:bd:ad:27:7b:62:fe:3a:da:93:8c:83:d5:66:
                    b1:00:9f:5e:55:9c:ca:f9:b2:85:b8:73:7a:71:3a:
                    57:86:ef:a5:ba:2f:16:6d:e6:62:75:3e:8f:6f:59:
                    4c:35:00:13:2e:af:a4:0d:f0:7d:38:81:fe:ea:b8:
                    8d:59:46:88:7a:8a:18:51:88:3a:f6:74:92:c7:27:
                    37:5c:9b:a2:50:86:44:ef:46:92:be:24:11:75:6c:
```

```
                    3e:bb
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Key Identifier:
                F0:9F:41:A4:9F:6B:38:F3:77:69:AF:C6:F5:58:D3:64:3F:9A:1F:AD
            X509v3 Authority Key Identifier:
                keyid:F3:71:14:32:42:C9:17:EA:65:5C:A8:56:FA:3C:9D:7C:05:B0:AD:44
                DirName:/C=DE/ST=Germany/L=Munich/O=TUM/OU=I8/CN=DS2OS CA
                serial:01:62:BF:67:0E:B1

            X509v3 Key Usage: critical
                Digital Signature, Non Repudiation, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
            Netscape Cert Type:
                SSL Client
            Netscape Comment:
                DS2OS service certificate for temperature
            1.3.6.1.4.1.1:
                ..sha256:temperatureManifest
            1.3.6.1.4.1.0:
                ..FALSE
            1.3.6.1.4.1.2:
                ..temperature
    Signature Algorithm: sha256WithRSAEncryption
         08:0b:e7:6c:bc:23:36:9e:c7:f4:56:07:d8:21:42:eb:2d:b0:
         c2:d8:1a:27:ea:76:7f:da:96:d2:1e:33:d3:b0:4e:37:62:f7:
         28:cc:04:0d:3c:32:51:40:ed:4b:1d:a1:7b:e1:a9:67:f1:0c:
         a5:69:29:b1:8c:b7:87:64:cd:ee:b6:00:c6:27:22:2e:5f:b4:
         c5:3c:f0:51:20:db:b2:b6:a1:cd:91:fc:4b:cd:d8:d0:1a:c6:
         4d:63:27:50:9d:ca:b1:59:82:8f:d5:a9:d3:2e:88:29:73:a0:
         a9:c3:93:dd:72:5a:09:24:19:24:cf:91:f7:a6:60:20:cd:31:
         17:de:fd:9b:94:65:09:69:94:d9:3b:95:3e:47:92:57:28:89:
         2d:2c:41:88:9f:57:99:36:5f:0a:10:88:15:82:9f:45:36:f4:
         2b:d9:34:9a:29:32:01:1b:d3:69:fb:22:ae:05:08:81:e7:c5:
         b4:74:a1:1f:fe:e4:02:ce:86:1c:ca:96:79:93:45:59:02:fc:
         a2:f7:77:fb:83:a0:6f:77:de:e5:55:99:69:aa:c8:e0:b0:bd:
         3c:75:68:7c:c2:e9:5e:bd:b2:04:13:35:cf:a6:74:d6:c7:b6:
         1a:ff:d5:41:37:02:17:ff:cb:26:f2:90:85:d9:f5:26:78:7c:
         c8:9e:b6:a0
-----BEGIN CERTIFICATE-----
MIIExTCCA62gAwIBAgIGAWUloULxMA0GCSqGSIb3DQEBCwUAMF4xCzAJBgNVBAYT
AkRFMRAwDgYDVQQIDAdHZXJtYW55MQ8wDQYDVQQHDAZNdW5pY2gxDDAKBgNVBAoM
A1RVTTELMAkGA1UECwwCSTgxETAPBgNVBAMMCERTMk9TIENBMB4XDTE4MDgxMDIw
NTczMFoXDTE4MDgxMDIxMDczMFowYTELMAkGA1UEBhMCREUxEDAOBgNVBAgMB0dl
cm1hbnkxDzANBgNVBAcMBk11bmljaDEMMAoGA1UECgwDVFVNMQswCQYDVQQLDAJJ
ODEUMBIGA1UEAwwLdGVtcGVyYXR1cmUwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAw
ggEKAoIBAQCzLF+7z3+Yw+oZAg/sgNcOSwk1Pjl/eVDWQg2C/zJAKEX4xs99Lsbn
HErKUEc21x8ltb33wguGA2nzpgmV9yVzHGi9Hs66Jk7w1OYyEmLcKvcVLsyXDG5D
cxVPbwRDZTUqkRRussDK8xpm2fhMpujwTVvMEmwRwwSnNecLq9vdkEGGctO5v08E
M/HG3C/8trwHSMN2NLFdnLHPmlvprvWF1V+GzNYdva0ne2L+OtqTjIPVZrEAn15V
nMr5soW4c3pxOleG76W6LxZt5mJ1Po9vWUw1ABMur6QN8HO4gf7quI1ZRoh6ihhR
```

```
iDr2dJLHJzdcm6JQhkTvRpK+JBF1bD67AgMBAAGjggGEMIIBgDAJBgNVHRMEAjAA
MB0GA1UdDgQWBBTwn0Gkn2s483dpr8b1WNNkP5ofrTCBjQYDVR0jBIGFMIGCgBTz
cRQyQskX6mVcqFb6PJ18BbCtRKFipGAwXjELMAkGA1UEBhMCREUxEDAOBgNVBAgM
B0dlcm1hbnkxDzANBgNVBAcMBk11bmljaDEMMAoGA1UECgwDVFVNMQswCQYDVQQL
DAJJ0DERMA8GA1UEAwwIRFMyT1MgQ0GCBgFiv2c0sTA0BgNVHQ8BAf8EBAMCBeAw
EwYDVR0lBAwwCgYIKwYBBQUHAwIwEQYJYIZIAYb4QgEBBAQDAgeAMDgGCWCGSAGG
+EIBDQQrFilEUzJPUyBzZXJ2aWNlIGNlcnRpZmljYXRlIGZvciB0ZW1wZXJhdHVy
ZTAmBgYrBgEEAQEEHAwac2hhMjU2OnRlbXBlcmF0dXJlTWFuaWZlc3QwEQYGKwYB
BAEABAcMBUZBTFNFMBcGBisGAQQBAgQNDAt0ZW1wZXJhdHVyZTANBgkqhkiG9w0B
AQsFAAOCAQEACAvnbLwjNp7H9FYH2CFC6y2wwtgaJ+p2f9qW0h4z07B0N2L3KMwE
DTwyUUDtSx2he+GpZ/EMpWkpsYy3h2TN7rYAxiciLl+0xTzwUSDbsrahzZH8S83Y
0BrGTWMnUJ3KsVmCj9Wp0y6IKX0gqc0T3XJaCSQZJM+R96ZgIM0xF979m5RlCWmU
2TuVPkeSVyiJLSxBiJ9XmTZfChCIFYKfRTb0K9k0mikyARvTafsirgUIgefFtHSh
H/7kAs6GHMqWeZNFWQL8ovd3+40gb3fe5VWZaarI4LC9PHVofMLpXr2yBBM1z6Z0
1se2Gv/VQTcCF//LJvKQhdn1Jnh8yJ62oA==
-----END CERTIFICATE-----
```

Listing 10: X.509 Certificate format of a temperature service

Chapter B

List of acronyms

AES	Advanced Encryption Standard.
AMQP	Advanced Message Queuing Protocol.
API	Application Programming Interface.
APK	Android PacKage.
ARM	Advanced RISC Machine.
BLE	Bluetooth Low Energy.
CA	Certificate Authority.
CMR	Context Model Repository.
COAP	Constrained Application Protocol.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
CRG	Certificate Revocation Guard.
CRL	Certificate Revocation List.
CSR	Certificate Signing Request.
deb	Debian package.
DoS	Denial of Service.
DS2OS	Distributed Smart Space Orchestration System.
DTLS	Datagram Transport Layer Secure.
E2E	End-To-End.
EM	Electromagnetic.

HAL	Hardware Abstraction Layer.
HTTP	HyperText Transfer Protocol.
IC	Integrated Circuit.
IETF	Internet Engineering Task Force.
IFTTT	If This Then That.
I/O	Input/Output.
IoT	Internet of Things.
IP	Internet Protocol.
ipa	IOS App Store Package.
ISO	International Organization for Standardization.
IT	Information Technology.
JVM	Java Virtual Machine.
JWT	JSON Web Token.
KA	Knowledge Agent.
KB	Kilobytes.
M2M	Machine to machine.
MAC	Medium access control.
MB	Megabytes.
MCU	Micro-Controller Unit.
MiTM	Man in The Middle.
MMU	Memory Management Unit.
MQTT	Message Queuing Telemetry Transport.
μ-service	Micro-service.
NLSM	Node Local Service Manager.
NTP	Network Time Protocol.
OCSP	Online Certificate Status Protocol.
OID	Object Identifier.
OS	Operating System.
OSI	Open Systems Interconnection. Reference model for layered network architectures by the OSI.
P2P	Peer-To-Peer.
PDU	Protocol data unit. Refers to a message at a specific layer of the OSI model including all headers and trailers of the respective layer and all layers above.

PKI	Public Key Infrastructure.
PoP	Proof-of-Possession.
PRNG	Pseudo-Random Number Generator.
RAM	Random Access Memory.
REST	Representational state transfer.
ROM	Read-Only Memory.
RTE	Runtime Environment.
S2Store	Smart Space Store.
SBC	Single-Board Computer.
SCTP	Stream Control Transmission Protocol. Datagram-oriented, semi-reliable transport layer protocol.
SDK	Software Development Kit.
SDU	Service data unit. Refers to the payload of a message at a specific layer of the OSI model excluding all headers and trailers of the respective layer.
SHA	Secure Hash Algorithm.
SHE	Service Hosting Environment.
SLCA	Site Local Certificate Authority.
SLSM	Site Local Service Manager.
SOA	Service Oriented Architecture.
SSH	Secure Shell.
SSL	Secure Sockets Layer.
TCP	Transmission control protocol. Stream-oriented, reliable, transport layer protocol.
TLS	Transport Layer Secure.
UDP	User datagram protocol. Datagram-oriented, unreliable transport layer protocol.
UI	User Interface.
URI	Unified Resource Identifier.
USB	Universal Serial Bus.
VSL	Virtual State Layer.
XML	EXtensible Markup Language.

Bibliography

[1] Kutalmis Akpinar and Kien A Hua. “ThingStore - An Internet of Things Management System.” In: *BigMM* (2017), pp. 354–361.

[2] OSGi Alliance. *OSGi*. `https://www.osgi.org/developer/architecture/`.

[3] Arwa Alrawais et al. “Fog Computing for the Internet of Things - Security and Privacy Issues.” In: *IEEE Internet Computing* 21.2 (2017), pp. 34–42.

[4] Ayan Mukherjee Amitranjan Gantait Joy Patra. *Securing IoT devices and gateways*. `https://www.ibm.com/developerworks/library/iot-trs-secure-iot-solutions1/iot-trs-secure-iot-solutions1-pdf.pdf`.

[5] Andrea, Ioannis, Chrysostomou, Chrysostomos, and Hadjichristofi, George C. “Internet of Things - Security vulnerabilities and challenges.” In: *ISCC* (2015), pp. 180–187.

[6] *Android Platform Architecture*. `https://developer.android.com/guide/platform/`. Google.

[7] *Android security white paper*. `https://static.googleusercontent.com/media/www.google.co.il/iw/IL/work/android/files/android-for-work-security-white-paper.pdf`. Google, May 2015.

[8] Auth0. *Introduction to JSON Web Tokens*. `https://jwt.io/introduction/`.

[9] Victoria Beltran and Antonio F Skarmeta. “An overview on delegated authorization for CoAP - Authentication and authorization for Constrained Environments (ACE).” In: *WF-IoT* (2016), pp. 706–710.

[10] Adhitya Bhawiyuga, Mahendra Data, and Andri Warda. “Architectural design of token based authentication of MQTT protocol in constrained IoT device”. In: *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*. IEEE, 2017, pp. 1–4.

[11] Bjorn Butzin, Frank Golatowski, and Dirk Timmermann. “Microservices approach for the internet of things”. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–6.

[12] Deniz Celik. "Semi-Autonomous IoT Service Management on Unattended Nodes". MA thesis. Technical University of Munich, Aug. 2018.

[13] Next Thing Co. *CHIP*. `http://nextthing.co/pages/chip`.

[14] Onion Corporation. *Omega2+*. `https://onion.io/store/omega2p/`.

[15] Ed. D. Hardt. *The OAuth 2.0 Authorization Framework*. Tech. rep. Oct. 2012. URL: `https://tools.ietf.org/html/rfc6749`.

[16] Digitsole. *Smartshoe*. `https://www.digitsole.com/`.

[17] Colin Dixon et al. "An Operating System for the Home." In: *NSDI* (2012).

[18] Shamini Emerson et al. "An OAuth based authentication mechanism for IoT networks". In: *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2015, pp. 1072–1074.

[19] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. "Security Analysis of Emerging Smart Home Applications." In: *IEEE Symposium on Security and Privacy* (2016), pp. 636–654.

[20] Polk W Ford V and Solo D. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. Tech. rep. Jan. 1999. URL: `https://tools.ietf.org/html/rfc2459`.

[21] Cloud Foundry Foundation. *Blue-Green Deployment*. `https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html`.

[22] OWASP Foundation. *Access Control*. `https://www.owasp.org/index.php/Category:Access_Control`.

[23] OWASP Foundation. *Integrity*. `https://www.owasp.org/index.php/Integrity`.

[24] OWASP Foundation. *Top IoT Vulnerabilities*. `https://www.owasp.org/index.php/Top_IoT_Vulnerabilities`.

[25] Raspberry Pi Foundation. *Raspberry Pi 3 Model B*. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`.

[26] Raspberry Pi Foundation. *Raspberry Pi Zero W*. https://www.raspberrypi.org/products/raspberry-pi-zero-w/.

[27] GlobalSign. *What is a Certificate Signing Request?* `https://www.globalsign.com/en/blog/what-is-a-certificate-signing-request-csr/`.

[28] Google. *Analyze your build with APK Analyzer*. `https://developer.android.com/studio/build/apk-analyzer`. Updated June 5, 2018.

[29] Google. *Google Cloud IoT*. `https://cloud.google.com/solutions/iot/`.

[30] *Guide to OCSP Stapling*. Tech. rep. Nov. 2013.

[31] R. Canetti H. Krawczyk M. Bellare. *HMAC: Keyed-Hashing for Message Authentication*. Tech. rep. Feb. 1997. URL: `https://tools.ietf.org/html/rfc2104`.

[32] T Hansen. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. Tech. rep. July 2006. URL: `https://tools.ietf.org/html/rfc4634`.

[33] Qinwen Hu, Muhammad Rizwan Asghar, and Nevil Brownlee. "Certificate Revocation Guard (CRG): An Efficient Mechanism for Checking Certificate Revocation". In: *2016 IEEE 41st Conference on Local Computer Networks (LCN)*. IEEE, 2016, pp. 527–530.

[34] *IANA*. `https://www.iana.org/`.

[35] *IFTTT*. `https://ifttt.com/`.

[36] Apple Inc. *Bundle Structures*. `https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html#/apple_ref/doc/uid/10000123i-CH101-SW1`. Updated 2017-03-27.

[37] Bouncy Castle Inc. *Bouncy Castle Crypto API*. `http://www.bouncycastle.org/index.html`.

[38] Docker Inc. *Docker*. `https://www.docker.com/`.

[39] Rancher Labs Inc. *RancherOS*. `https://rancher.com/rancher-os/`.

[40] Red Hat Inc. *Standard X.509 v3 Certificate Extension Reference*. `https://access.redhat.com/documentation/en-US/Red_Hat_Certificate_System/8.0/html/Admin_Guide/Standard_X.509_v3_Certificate_Extensions.html`.

[41] LLC Independent Security Evaluators. *IoT Village*. `https://www.iotvillage.org/`.

[42] *iOS Security Guide*. `https://www.apple.com/lae/ios/app-store/`. Apple Inc, Jan. 2018.

[43] P Jones. *US Secure Hash Algorithm 1 (SHA1)*. Tech. rep. Sept. 2001. URL: `https://www.ietf.org/rfc/rfc3174.txt`.

[44] Sahiti Kappagantula. *Microservice Architecture*. `https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a`.

[45] Hamzeh Khazaei, Hadi Bannazadeh, and Alberto Leon-Garcia. "SAVI-IoT - A Self-Managing Containerized IoT Platform." In: *FiCloud* (2017), pp. 227–234.

[46] Hokeun Kim et al. "A Toolkit for Construction of Authorization Service Infrastructure for the Internet of Things." In: *IoTDI* (2017), pp. 147–158.

[47] Jaeho Kim and Jang-Won Lee. "OpenIoT - An open service framework for the Internet of Things." In: *WF-IoT* (2014), pp. 89–93.

[48] Swati Kinikar and Sujatha Terdal. "Implementation of open authentication protocol for IoT based application". In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, 2016, pp. 1–4.

[49] N. Sakimura M. Jones J. Bradley. *JSON Web Token (JWT)*. Tech. rep. May 2015. URL: `https://tools.ietf.org/html/rfc7519`.

[50] Patrick D McDaniel and Aviel D Rubin. “A Response to ”Can We Eliminate Certificate Revocation Lists?”.” In: *Financial Cryptography* 1962.Chapter 17 (2000), pp. 245–258.

[51] Microsoft. *Azure IoT Hub*. `https://azure.microsoft.com/en-us/services/iot-hub/`.

[52] Tal Mizrahi and Danny Mayer. *Network Time Protocol Version 4 (NTPv4) Extension Fields*. Tech. rep. Mar. 2016. URL: `https://tools.ietf.org/html/rfc7822`.

[53] Arsalan Mosenia and Niraj K Jha. “A Comprehensive Study of Security of Internet-of-Things.” In: *IEEE Trans. Emerging Topics Comput.* 5.4 (2017), pp. 586–602.

[54] *MQTT*. `http://mqtt.org/`.

[55] MyMotiv. *Motiv Ring*. `https://mymotiv.com/`.

[56] O2. *How Mobile Phones Changed Your World*. `https://www.telegraph.co.uk/technology/how-phones-changed-the-world/evolution-of-mobile-phones/`.

[57] OASIS. *AMQP*. `https://www.amqp.org/`.

[58] Fabian Ohlenforst. “Providing a Remotely Manageable Runtime Environment for IoT Services”. MA thesis. Technical University of Munich, Aug. 2018.

[59] M.-O. Pahl. “Distributed smart space orchestration”. PhD thesis. Technical University of Munich, 2014.

[60] Mukul Panwar and Ajay Kumar. “Security for IoT: An effective DTLS with public certificates”. In: *2015 International Conference on Advances in Computer Engineering and Applications (ICACEA)*. IEEE, 2015, pp. 163–166.

[61] Chang-Seop Park. “A Secure and Efficient ECQV Implicit Certificate Issuance Protocol for the Internet of Things Applications”. In: *IEEE Sensors Journal* 17.7 (2017), pp. 2215–2223.

[62] GnuPG Project. *GnuPG*. `https://www.gnupg.org/`.

[63] *Public Key Infrastructure(X.509) (pkix)*. `https://datatracker.ietf.org/wg/pkix/about/`.

[64] Chris Richardson. *Pattern: Microservice Architecture*. `http://microservices.io/patterns/microservices.html`.

[65] R Rivest. *The MD5 Message-Digest Algorithm*. Tech. rep. Apr. 1992. URL: `https://www.ietf.org/rfc/rfc1321.txt`.

[66] Ronald L Rivest. “Can We Eliminate Certificate Revocations Lists?” In: *Financial Cryptography* 1465.Chapter 14 (1998), pp. 178–183.

[67] Stefan Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. Tech. rep. June 2013. URL: `https://tools.ietf.org/html/rfc6960`.

[68] Danilo Sato. *CanaryRelease*. https://martinfowler.com/bliki/CanaryRelease.html.

[69] Michael Schukat and Pablo Cortijo. "Public key infrastructures and digital certificates for the Internet of things". In: *2015 26th Irish Signals and Systems Conference (ISSC)*. IEEE, 2015, pp. 1–5.

[70] Savio Sciancalepore et al. "OAuth-IoT - An access control framework for the Internet of Things based on open standards." In: *ISCC* (2017), pp. 676–681.

[71] *Security Checklist for the Internet of Things*. Particle, Jan. 2017.

[72] Amazon Web Services. *AWS IoT*. https://aws.amazon.com/iot/.

[73] *Snapcraft*. https://snapcraft.io/. Canonical.

[74] Statista. *IoT connected devices installed base worldwide from 2015 to 2025*. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[75] STMicroelectronics. *STM32F4 Series*. https://www.st.com/en/microcontrollers/stm32f4-series.html?querycriteria=productId=SS1577.

[76] VoCore Studio. *VoCore2*. "http://vocore.io/v2.html".

[77] Inc Sun Microsystems. *Netscape-Defined Certificate Extensions*. https://docs.oracle.com/cd/E19957-01/816-5533-10/ext.htm.

[78] *The Internet of Things Reference Model*. Cisco, June 2014. URL: http://cdn.iotwf.com/resources/71/IoT_Reference_Model_White_Paper_June_4_2014.pdf.

[79] Emin Topalovic et al. In: (2012).

[80] *Ubuntu Core*. https://www.ubuntu.com/core. Canonical.

[81] *Ubuntu Core 16 - Security*. Version 2.0.0 rc9. Canonical, Aug. 2017.

[82] *uClinux*. http://www.uclinux.org/index.html.

[83] *USB Tester Spannungsprüfer UM25C*. https://www.amazon.de/Spannungspr%C3%BCfer-Voltmeter-USB-Digital-Multimeter-Tester-Amperemeter-Kommunikation/dp/B07C8CD7QX.

[84] PK Vitality. *K'Track Glucose*. https://www.pkvitality.com/ktrack-glucose/.

[85] Hongwei Zhang. *Clock Synchronization in Distributed Systems Using NTP and PTP*. http://www.cs.wayne.edu/~hzhang/courses/8260/Lectures/Chapter%2021%20-%20Clock%20Synchronization%20in%20Distributed%20Systems%20Using%20NTP%20and%20PTP.pdf.