Securing the Device Lifecycle Management: A Scalable and Cost-Efficient Public Key Infrastructure Through Microservices

Sara Sumaidaa[1], Hamda Almenhali[1], Rajkumar Ramasamy[1], Oleksii Voronin[2], Mohammed Alazzani[1] and Kyusuk Han[1]

[1]*Technology Innovation Institute, U.A.E.*

[2]*Unikie MEA Ltd., U.A.E.*

{*sara.awadh, hamda.almenhali, rajkumar.ramasamy, oleksii.voronin, mohammed.alazzani, kyusuk.han*}*@tii.ae*

Keywords: Device Lifecycle Management, Infrastructure Security, Microservice, PKI, Provisioning, Software Update.

Abstract: Ensuring the secure operation of infrastructure and devices throughout their lifecycle is crucial. This includes secure key provisioning, certificate management, and software updates, all essential for effective device lifecycle management. Despite the development and deployment of numerous architectures, minimizing the financial strain associated with their administration remains a challenge. Although cloud-based approaches are widely adopted, certain environments, such as industrial plants, cannot fully benefit from these solutions due to limited network connectivity. Establishing connectivity for Public Key Infrastructure (PKI) or over-the-air software updates in such settings can be particularly complex due to stringent security requirements. To address this challenge, we propose a cost-effective solution using a microservice model to consolidate software management, certificate management, and key provisioning in a single centralized location. This approach is well-suited for environments with limited network connectivity. By adopting this framework, we ensure scalability, flexibility, and streamlined management, providing an efficient solution to manage devices in diverse environments.

1 INTRODUCTION

Ensuring the security of a device is an ongoing process rather than a one-time effort. Although a device may initially be secure, it remains vulnerable to emerging threats throughout its operational lifespan. Device Lifecycle Management (DLM) encompasses a comprehensive approach to safeguarding a device from acquisition through to disposal.

Extensive research (Miettinen et al., 2018; Soos et al., 2018; Gaudio et al., 2020; Howell et al., 2023) has been conducted on various aspects of DLM. Effective DLM requires the implementation of various security functions at different stages to maintain the operational integrity of devices throughout their lifecycle. For instance, key and credential management is a critical security function as it facilitates authentication and secure communications. This includes key provisioning, key management, and software updates.

Despite the imperative deployment of diverse security infrastructures, their implementation and maintenance pose significant challenges, particularly in terms of cost and complexity. Establishing the necessary hardware infrastructure, along with the ongoing expenses for software upgrades and security patches throughout the device's lifespan, can be especially onerous for startups or small businesses with limited financial resources.

To address these challenges, various approaches have been proposed. For instance, cloud-based PKI has evolved beyond a mere research concept (Huang et al., 2015); major providers such as Amazon, Google, and Microsoft now offer robust PKI solutions in the cloud. These services mitigate the need for substantial upfront investments in software or hardware licenses. Additionally, several researchers (Dua et al., 2020; Schaerer et al., 2022) have proposed blockchain-based distributed PKI infrastructures. However, concerns persist regarding the reliance on connected environments, particularly in settings such as factory facilities, where access to cloud services may be restricted due to heightened security risks during the manufacturing stage, when devices are particularly vulnerable.

In this paper, we present a design that provides essential functions such as provisioning, certificate management, and software management, specifically tailored for small-scale environments, utilizing mi-

Sumaidaa, S., Almenhali, H., Ramasamy, R., Voronin, O., Alazzani, M. and Han, K.
Securing the Device Lifecycle Management: A Scalable and Cost-Efficient Public Key Infrastructure Through Microservices.
DOI: 10.5220/0013171700003899

In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 2*, pages 342-352
ISBN: 978-989-758-735-1; ISSN: 2184-4356

croservice architectures (Hasselbring, 2016; Rossi et al., 2020). We address the challenge of maintaining security within a unified system and propose a novel, cost-effective approach to enabling these functions. We also present a case study demonstrating the implementation of this design using several commercially off-the-shelf (COTS) solutions. By integrating these components into a microservice architecture, we achieve rapid development and deployment of key management functions, including key provisioning, software update management, and certificate management, while preserving security and providing flexibility and scalability for future improvements. Furthermore, we demonstrate how to facilitate the transition from small-scale information systems to large-scale infrastructures by enabling the independent deployment and scaling of microservices.

The structure of this paper is as follows: Section 2 and 3 discuss the problem definitions in Microservice based Key Management Infrastructures. Section 4 presents our proposed model. Section 5 shows the evaluation results with a case study of the design implementation. Section 6 discusses the related work and Section 7 concludes the paper.

2 KEY MANAGEMENT IN IoT DEVICE LIFECYCLE

The concept of IoT Device Lifecycle Management (IoT DLM) originates from the critical necessity to ensure the operability of IoT devices throughout their lifecycle. Although different terminologies are used (Miettinen et al., 2018; Soos et al., 2018; Gaudio et al., 2020; Howell et al., 2023), they generally encompass the stages from deployment to end-of-life, which we categorize into **Deployment**, **Operation**, and **End-of-Life** in this section.

Deployment represents the initial stage, involving the *Manufacturing* or *Production* of devices, followed by *Provisioning* and *Configuration* to prepare devices for deployment. **Operation** encompasses the *Maintenance*, *Monitoring*, and *Software Updates* required to ensure devices remain functional and secure. **End-of-Life** is the final stage, involving the *Decommissioning* or *Secure Disposal* of the devices once they reach the end of their lifecycle.

Ensuring device security throughout its lifecycle is paramount, necessitating a holistic approach that provides security functions across all stages. For example, the National Institute of Standards and Technology (NIST) (Howell et al., 2023) publishes guidelines to secure the DLM of IoT in enterprises. Researchers have repeatedly proposed enhancements to these functions to enhance the security of critical infrastructures. For example, numerous approaches to improve PKI have been suggested, such as distributed PKI mechanisms (Schaerer et al., 2022) and hybrid mechanisms incorporating identity-based cryptography into PKI (Tan et al., 2015; Chia et al., 2021). Software updates are crucial to keeping devices current and secure (Kuppusamy et al., 2017; Al Blooshi and Han, 2022), while provisioning methods are actively being studied (Kwon et al., 2018; Kohnhäuser et al., 2021).

From a key and credential management perspective, it is essential to provide appropriate key management infrastructures tailored to each stage of the IoT device lifecycle. As depicted in Fig.1, we identify the security functions related to key management throughout the lifecycle.

2.1 Key Provisioning in Deployment

When a device is manufactured, it may not contain any secret to establish an initial secure operation yet. To enable secure communication, provisioning the secret keys or credentials to the blank device is required. **Provisioning Function** is a critical process that provisions cryptographic keys to devices during the deployment stage. Usually, key provisioning is performed in the factory, which is disconnected from the network.

2.2 Software Updates in Operation

Vulnerabilities in software could affect the security of key management. Resolving known vulnerabilities in a timely manner is a critical requirement to enable secure key management. **Software Update Functions** ensure the device remains up-to-date throughout its operation.

2.3 Certificate Management Across all Stages

Certificate management is a function used throughout the device's lifecycle, from enrollment to revocation of devices. This function also involves key provisioning. Devices may request certificate updates over the network.

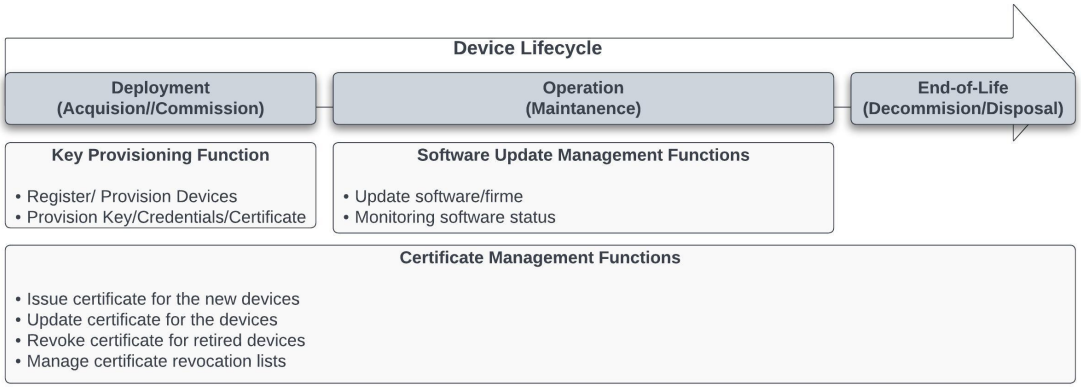

Figure 1: Key Management Functions in IoT Device Lifecycle.

3 INFRASTRUCTURE MANAGEMENT ISSUES

It is imperative for every organization to consider the deployment of security functions. However, implementing them as a large-scale, on-premise infrastructure can significantly increase the burden of deployment and management. Consequently, the primary objective of this work is to design a flexible, secure and cost-effective model for managing essential functions, such as provisioning, certificate management, and software updates, within a small-scale infrastructure. At the same time, the implemented system should be deployable or migratable to a large-scale distributed environment without design modification as we define a requirement **Req.1** in Section 3.2.

Moreover, the system must account for the different environmental conditions under which each function operates. *Provisioning functions* are executed in factory or manufacturing environments at an early stage, where the device may lack the capability to verify any injected input. Consequently, *provisioning functions* are conducted in physically protected environments, with external network connections prohibited. In contrast, *certificate management* necessitates communication to issue updated certificates, revoke existing ones, or generate certificate revocation lists, thereby supporting devices throughout their entire lifecycle. Similarly, *software update functions* require connectivity to external software repositories to monitor and download new updates. We define a requirement **Req.2** in Section 3.2.

3.1 Security Considerations

In deploying multiple security functions, we discuss the potential threat scenarios that must be considered for the system providing key provisioning, certificate management, and software updates.

3.1.1 Threats in Key Provisioning

During the initial provisioning phase, the device lacks the necessary keys or credentials to verify any data. This vulnerability presents a significant security risk, as an attacker could potentially install compromised keys or software, thereby gaining control over the device. Although we assume that provisioning occurs in a physically isolated environment and is handled by authorized personnel, remote access attacks remain a concern. Therefore, we define the requirement **Req.3** in Section 3.2.

3.1.2 Threats in Certificate Management

In certificate management, an attacker may attempt to issue certificates for their own devices by sending a certificate signing request (CSR) that pretends to be from a legitimate blank device. Once the certificate is issued, the attacker's device can masquerade as a genuine device. To mitigate this risk, we define the requirement **Req.4** in Section 3.2.

By addressing these challenges with the proposed solutions, the model aims to provide a secure and flexible framework to efficiently manage critical infrastructure functions.

3.1.3 Threats in Software Updates

During the operation of software update management, the system retrieves new software or firmware from the repository for installation on the device. If malicious software is installed, the device can be compromised, allowing an attacker to gain access and control. Malicious software could originate from a harmful repository, be modified or compromised by an attacker, or be an outdated version of the software.

Moreover, an attacker could attempt to compromise the software while it is stored locally within the system, waiting to be installed on the device. It is important to note that software can be transmitted over the network or delivered through out-of-band channels. Thus, we define the requirement **Req.5** in Section 3.2.

According to (Kuppusamy et al., 2017), revealing the current version of the software to any malicious entity also increases the risk that an attacker could exploit the vulnerabilities of the system. Thus, we explicitly include the **Req.6** in Section 3.2.

3.2 Requirements

We define the requirements based on the issues identified in this section.

Req.1: The system shall maximize cost efficiency in providing security functions while ensuring scalability to accommodate growth.

Req.2: The system shall provide appropriate security functions to satisfy their operational conditions.

Req.3: The system shall only allow physical access during the provisioning phase.

Req.4: The Certificate Authority shall issue certificates for devices only through an authorized entity.

Req.5: The system shall maintain the integrity of the software until performing the installation on the device.

Req.6: The system shall prevent the leakage of information about the current software version in the device to unauthorized entities.

4 MICROSERVICE-BASED KEY MANAGEMENT ARCHITECTURE

In this section, we propose the design of a system that integrates software management, certificate management, and key provisioning functions. This system is capable of operating on a single physical unit while also being scalable for deployment in large-scale infrastructures.

4.1 System Architecture

Figure 2 depicts our system architecture, communicating with external entities, including *Remote Repository* and *devices*. By employing a microservice-based architecture, each module is isolated as a separate microservice.

4.1.1 Provisioning Module

The primary role of **P**rovisioning **M**odule (*pm*) is to install the software and provision keys to the target device at the provisioning stage. The *pm* contains the public key of the software management module (pk_{sm}) and the certificate management module(pk_{cm}). *pm* manages its public key pair and public keys of *km* and *sm*. The *pm* operates in two steps:

Step 1: Provisioning Device Software and CA Certificates. Installing the root and associated CA certificates, along with the device software, is the first step in the provisioning process. Once Step 1 is completed, the device can proceed to Step 2.

Step 2: Issuing Certificate of the Device Public Key. The second step involves issuing a certificate for the device's public key. The device generates its public key and requests the issuance of a certificate for this key from the key management system *km* through *pm*.

4.1.2 Certificate Management Module

The Certificate (and **K**ey) **M**anagement module (*km*) handles the crucial tasks of managing certificates and keys. The key management system *km* encompasses the roles of the *Certificate Authority* (CA), the *Registration Authority* (RA), and the *Validation Authority* (VA). *km* manages its public key pair and public keys of *sm* and *pm*.

4.1.3 Software Management Module

The **S**oftware **M**anagement module (*sm*) is responsible for overseeing all software management functions, including the distribution of software updates and the management of software package storage and deployment. The system ensures that only authenticated and validated updates are delivered to the device. *sm* manages its public key pair and public keys of *km* and *pm*.

4.1.4 Communication Module

The Communication Module (*cm*) facilitates interactions with external entities, including remote repositories or other devices via a wireless channel. Note that We omit the details of *cm* as it pertains to generic secure communication protocols, such as *TLS*.

4.1.5 External Entities

In our scenario, we define two types of external entities:

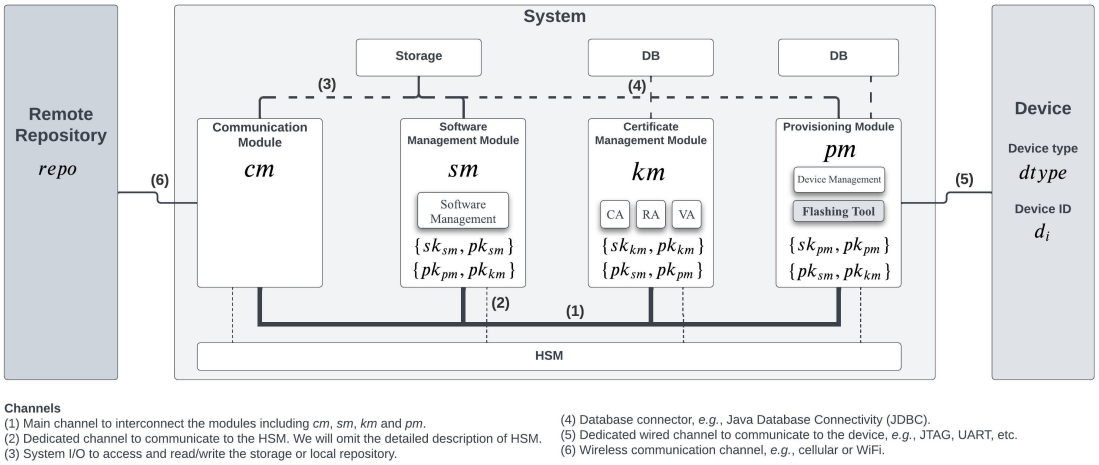

Figure 2: System Architecture of Provisioning System.

Remote Repository. The Remote Repository *repo* serves as an infrastructure to manage new software packages those are to be distributed to the system for installation on the devices. We consider the repository model from the infrastructure perspective in Uptane (Kuppusamy et al., 2017).

Device. Device refers to the target embedded hardware supported by the system in software management. The system manages specific device type *dtype* and n number of devices d_i, where $1 \leq i \leq n$.

4.2 Provisioning into the Device

Provisioning operations include two scenarios: one is *pm* flashes the software (*i.e.*, root certificate and other shared credentials), and the other is *pm* flashes the certificate of the device public key at the Provision stage. Since this operation shall be performed in physically protected environment, and *cm* must be turned off, and only necessary modules shall remain turned on as depicted in Fig. 3 and Fig. 4.

4.2.1 Scenario 1: Flashing Software to the Blank Device

The provisioning process begins by fetching the software from the local storage. Algorithm 1 and Fig. 3 depict the overall flows.

By provisioning software, the device now has public key of root CA and other necessary credential to interact with servers.

4.2.2 Scenario 2: Provisioning Certificate

For the certificate issuing during the provisioning, devices request the certificates to *km* through *pm*. Fig 4

Data: *pm* has pk_{repo} and pk_{sm}.;
$sw_{j+\alpha}$, $meta_{repo}^{j+\alpha}$, and $sig_{sm}^{j+\alpha}$ in the local repository
pm load $sw_{j+\alpha}$, $meta_{j+\alpha}$, and $sig_{sm}^{j+\alpha}$ from the local repository.;
pm verifies $sw_{j+\alpha}$ and $meta_{repo}^{j+\alpha}$ with $sig_{sm}^{j+\alpha}$ and pk_{sm}.;
if $sig_{sm}^{j+\alpha}$ *is valid* **then**
  Verify $sw_{j+\alpha}$ with $meta_{repo}^{j+\alpha}$ and pk_{repo}.;
  if $meta_{repo}^{j+\alpha}$ *is valid* **then**
    Proceed flashing $sw_{j+\alpha}$ to the device d_i
  end
  .
end

Algorithm 1: Provisioning Software.

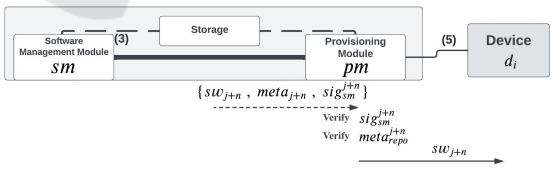

Figure 3: *pm* verifies new software $sw_{j+\alpha}$ then install $sw_{j+\alpha}$ to the device d_i.

describe the flow of issuing a certificate with *Certificate Signing Request* (CSR), which is standardized as PKCS#10.

For certain devices that are not capable of generating a CSR, we have also designed a solution using the *Certificate Request Message Format* (CRMF), as specified in RFC 4211. In this scenario, the device may send its public key to *pm* through the out-of-band channel, and *pm* generates the request on behalf of

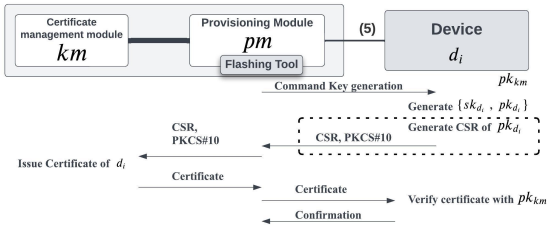

Figure 4: d_i request certificate with CSR. CSR is generated by d_i.

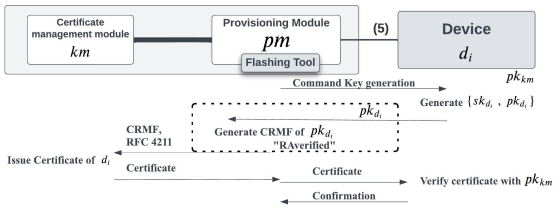

Figure 5: d_i request certificate with CRMF. CRMF can be generated by either d_i or *pm* with '*RAverified*.'.

the device, as depicted in Fig 5. Note that the '*RA verified*' stamp on the CRMF proof of possession indicates that the request has been successfully verified by the RA.

4.3 Certificate Management

Certificate management includes two cases. The first case is *certificate provisioning*, and the other case is *certificate update*.

4.3.1 Certificate Provisioning

As described in Section 4.2.2, *km* participates during the provisioning operation as *RA* and *CA*. When the new request with CSR or CRMF of d_i is received, *km* check the ownership of private key sk_{d_i}. We assume the environment that *km* and *pm* are running is physically protected, and the network connection stays offline, as *cm* is disabled.

4.3.2 Certificate Update

When it is not during the provisioning operation, *km* functions as the *Registration Authority* (RA), *Certificate Authority* (CA), and *Validation Authority* (VA). As shown in Fig. 6, certificate updates can occur with devices or other CAs depend on the implementation choice. Since *km*'s role mirrors that of a generic CA in a typical PKI scenario, we omit further details. During this process, *pm* is strictly disabled.

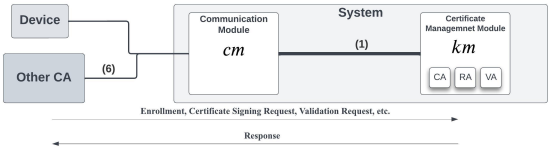

Figure 6: *km* becomes the generic CA when provisioning is not operated.

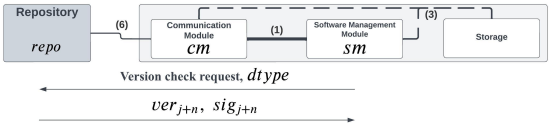

Figure 7: *sm* sends new software check request to Remote Repository.

4.4 Software Management

Software management operations include downloading software from the repository and storing it in the local system, tasks handled by the Software Management Module (*sm*) within the system.

Let *sm* is currently managing a software sw_j for the device model $dtype$ with the version number ver_j, where j stands for the count number of the current software version that is managed by *sm*.

sm may need to check if a new software $sw_{j+\alpha}$ is available in *repo*, where $\alpha \geq 1$, to keep the device up-to-date state.

4.4.1 Establishment of the Secured Connection to the Remote Repository

At first, *sm* connects to the remote repository *repo* through the communication module *cm*. We omit the detail since establishing the authenticated and encrypted channel could be done using the standard methods such as TLS.

4.4.2 Check Existence of the New Software

Checking the new software could be performed either by *sm* triggering the update check proactively as depicted in Fig.7, or by the repository *repo* periodically notifying that new update is available. In Fig. 7, the signature $sig_{j+\alpha}$ is $sig_{j+\alpha} = sign(sk_{repo}, ver_{j+\alpha}, dtype)$, where $sign(k,m)$ stands for the signing m with k.

4.4.3 Download the Software

When $sw_{j+\alpha}$ exists, *sm* initiates the process to download the updated software. Algorithm 2 shows the flow of downloading procedure. To minimize unnecessary data transfer and reduce bandwidth usage, the

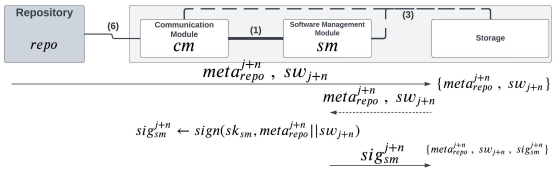

Figure 8: *sm* stores new software and the metadata with the signature.

system first downloads the metadata associated with the new software version.

Metadata $meta^{j}_{repo}$ essentially includes the information of *sw*, $mete^{j}_{repo} = \{data^{j}_{repo}, sig^{j}_{repo}\}$, where payload $data^{j}_{repo}$ including the software version ver_j, the hash of *sw* $hash_j$, where $hash_j = hash(sw_j)$, the size of *sw*, and sig^{j}_{repo}, which is a signature of $data^{j}_{repo}$, $sig^{j}_{repo} = sign(sk_{repo}, data^{j}_{repo})$.

Data: *sm* knows ver_j, *dtype*, and pk_{repo};
Received $ver_{j+\alpha}$ and $sig_{j+\alpha}$ from *repo*
Verify $ver_{j+\alpha}$ with $ver_{j+\alpha}$, *dtype* with pk_{repo};
if $sig_{j+\alpha}$ *is valid* **then**
  Compare $ver_{j+\alpha}$ to ver_j;
  if $ver_{j+\alpha} > ver_j$ **then**
    Download the metadata $meta^{j+\alpha}_{repo}$
  end
end
$meta_{j+\alpha}$ contains payload $data^{j+\alpha}_{repo}$ and the signature $sig^{j+\alpha}_{repo}$;
Verify $data^{j+\alpha}_{repo}$ and $sig^{j+\alpha}_{repo}$ with pk_{repo};
if $sig^{j+\alpha}_{repo}$ *is valid* **then**
  Download $sw_{j+\alpha}$;
  generate $hash(sw_{j+\alpha})$ and compare to $hash^{j+\alpha}$, extracted from $data^{j+\alpha}_{repo}$;
  if $hash^{j+\alpha} \equiv hash(sw_{j+\alpha})$ **then**
    Store $meta^{j+\alpha}_{repo}$ and $sw_{j+\alpha}$ at the local repository
  end
end

Algorithm 2: Download the new software.

4.4.4 Local Software Management

When *sm* stores $meta^{j+\alpha}_{repo}$ and $sw_{j+\alpha}$ to the local repository, *sm* could generate the signature of them $sig^{j+\alpha}_{sm}$, where $sig^{j+\alpha}_{sm} = sign(sk_{sm}, meta^{j+\alpha}_{repo}||sw_{j+\alpha})$ as depicted in Fig.8. To prevent the leakage of the metadata, the local repository could be managed by an access control mechanism or by storing $meta^{j+\alpha}_{repo}$ in an encrypted state.

4.4.5 Report the Update to the Server

After completing the download and storing the software in the local repository, or continuing with the provisioning process, the software management module *sm* may report to the repository *repo* that the download and verification have been successfully completed. This ensures a record is kept and prevents unnecessary redundancies. This step requires a customization of the Uptane model (Kuppusamy et al., 2017), which typically completes the process after the device has finalized the update.

5 EVALUATION

In this section, we evaluate our design that meets requirements in Section 5.2 and also show the performance upon scenarios in Section 5.4.

5.1 Design Analysis

We show that our design meet the requirements defined in Section 2.

5.1.1 Scalable Design with Cost Efficiency

As depicted in Figure 2, our design integrates the *Key Provisioning*, *Certificate Management*, and *Software Management* functions as microservices within a unified system. Leveraging the inherent advantages of a microservice-based architecture (Chandramouli, 2019; Ugale and Potgantwar, 2023; Oluyede et al., 2024; Pandiya, 2021), our design facilitates effective scalability.

Every function we deploy in our system is employed as the microservice and managable with minimum burden. For example, we utilized a COTS solution *EJBCA* for the key management function *km*, and replacing to another COTS solution is available. Our implemented *pm* is also changeable when the device is changed, additionally, by adding multiple *pm*s, multiple different types of devices can be supported. Fig. 9 depicts the case that CA and softwar update capability is located in different system, while connecting two provisioning servers supporting different device types. Thus, it is trivial to address that our design meets **Req.1**.

5.1.2 Support Different Operation Environment

Using microservice-based architecture, each function can be managed independently as addressed by (Chandramouli, 2019). In our system implementation, we used Docker to enable each module to be

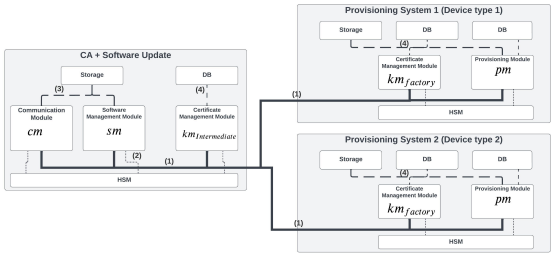

Figure 9: Example case: Physically separated functions.

Table 1: State of modules in each operation.

Operation	*cm*	*sm*	*km*	*pm*
Provisioning software	off	on (opt.)	off	on
Provisioning certificate	off	off	on	on
Software Management	on	on	off	off
Certificate Management	on	off	on	off

easily turned on and off based on the use cases. Table 1 shows the state of modules in each operation. For example, during the provisioning of a certificate, the communication module *cm* is turned off, and the remote access is not allowed. In contrast, when the provisioning module *pm* is turned off and the communication modeul *cm* is turned on to provide the network based certificate management while preventing malicious access to the blank devices. Thus, we claim **Req.2** holds.

5.2 Security Analysis

5.2.1 Security Against the Remote Attack During Provisioning

A remote attacker *Adv* may try to access the system while the provisioning is in progress. However, during this process, the communication module and other modules not involved in the provisioning operation are disabled, as addressed in Section 5.1.2. Thus, we claim **Req.3** holds.

5.2.2 Security Against Issuing Certificate to Unauthorized Devices

As we assume that only authorized personnel manages *pm* and the blank devices are managed in the physically protected environment. Thus, during provisioning stage, *km* and *pm* establish the secure connection, and sign messages between them. Since *pm* manages devices physically, issuing certificates to unauthorized entity can be prevented.

It is important to note that, in practice, a insider attacker *Adv* may compromised *km* or *pm* and attempt to issue certificates to malicious devices. Therefore, a strict security policy must be enforced to restrict system access by unauthorized personnel. Thus, we claim **Req.4** holds.

5.2.3 Security Against the Compromised Software Installation

Following the model presented in (Kuppusamy et al., 2017), our design specifically addresses scenarios where malicious software is received and pending installation.

Consider a remote attacker, denoted as *Adv*, who generates compromised software sw^{Adv}, its signature sig^{Adv}, and the associated metadata $meta^{Adv}$. The attacker must create $meta^{Adv}$ without access to the private key sk_{repo} and transmit it to *sm* via *cm*. The probability that a legitimate *sm* accepts $meta^{Adv}$ is no greater than the likelihood of breaking the employed public key cryptographic primitive.

Assume that sw^{Adv} is accepted by a compromised *sm*. Using sk_{sm}, *sm* generates a valid signature sig_{sm}^{Adv}, where $sig_{sm}^{Adv} \equiv \text{sign}(sk_{sm}, meta^{Adv}||sw^{Adv})$. Despite the validity of sig_{sm}^{Adv}, the signature of $meta^{Adv}$ becomes invalid, allowing *pm* to detect the compromise of *sm*. If sig_{sm}^{Adv} also becomes invalid, *pm* may consider the entire system compromised and halt all operations.

Instead of deploying malicious sw^{Adv}, a compromised *sm* might use sw_{j+1} for *pm*, even though sw_{j+2} is already available from the remote repository. However, since sw_{j+1} is still a higher version than the currently installed sw_j, the incentive for a successful attack remains low.

Finally, the device may verify the software's integrity post-installation using methods such as secure boot. Thus, we claim **Req.5** holds.

5.2.4 Security Against Leaking the Current Software Version

During checking the up-to-date software version, *sm* doesn't reveal the software version those currently installed to devices. Also, *sm* only proceed the software download after $sig_{j+\alpha}$ is verified. An attacker *Adv* may try to generate ver^{Adv} with slightly higher version number and sig^{Adv} without sk_{repo}. The probability of success that *Adv* generates the signature without private key is same as one of breaking the public key crypto primitive used.

Once the metadata *meta* is downloaded, it could be stored in the local repository with encrypted state, where the encryption key is only shared between *sm* and *pm*. As far as *sm* or *pm* are not compromised, the data will be stay encrypted. When the metadata is to be decrypted, communication to the external network is disabled, and only provisioning modules are active.

To avoid such risk, adding more authentic method to restrict the unauthorized access could be deployed. Thus, we claim **Req.6** holds.

5.3 Implementation Using COTS

In this section, we demonstrate the implementation of our design using Commercially Off-The-Shelf (COTS) products. We implemented all functions as containers on Docker. This approach allows for the easy disabling of unnecessary modules for specific operations.

For the *channel (1)* in Fig. 2, we adopted the *Neural Autonomic Transport System* (NATS)[1], which provides a messaging service for microservices, to facilitate communication between each container.

For the certificate management function *km*, we employed the *Enterprise Java Beans Certificate Authority* (EJBCA), a comprehensive certificate authority implementation by Keyfactor[2], which has been utilized in several research works (Tan et al., 2015; Jain et al., 2018; Rasyid et al., 2022). Using EJBCA WSDL/SOAP typed Web Service (EJBCA WS) APIs, we implemented an **interpreter** that translates these messages into JSON-type messages that can be exchanged over NATS.

Figure 10 illustrates an example where EJBCA receives a 'create certificate' (*createCertificate*) request from another entity through NATS. The initial message is in JSON format and is interpreted into a WSDL/SOAP message that EJBCA WS can process. In response, EJBCA issues a certificate based on the CSR and sends it back through NATS in the reverse order.

For the provisioning function *pm*, we implemented a versatile design that manages certificate handling during the provisioning process and supports various device types. We identified that the *flashing tool*, as depicted in Fig. 2, may depend on the specific device types. We tested with the *PolarFire SoC FPGA* from Microchip Inc. as a device type *dtype*, utilizing their proprietary software called *Libero Pro*. By containerizing this tool, *pm* can be easily replaced when a new device type is introduced.

For the software management function *sm*, we modeled the operation after the *primary ECU* in Uptane (Kuppusamy et al., 2017). Customization arises from scenarios where the software may not be immediately signed; hence, signing metadata from *repo* is included, and the report confirms that *sm* has successfully downloaded the software.

[1]https://nats.io

[2]https://www.keyfactor.com/

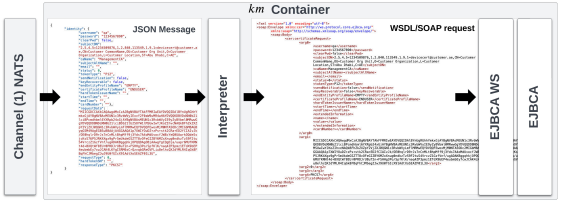

Figure 10: Connecting EJBCA to NATS: Other module connected to NATS transmits the request of 'createCertificate' message as JSON format, and interpreter in *km* convert JSON message to WSDL/SOAP message format that can run EJBCA WS.

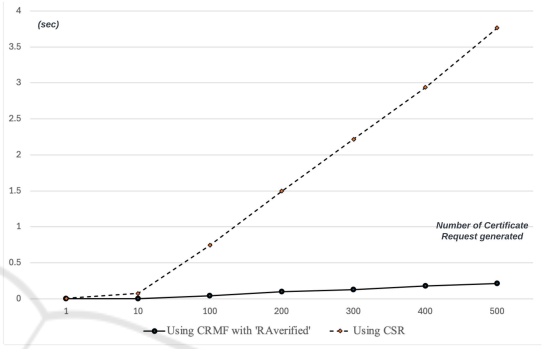

Figure 11: Time comparison between CSR generation in the device and CRMF generation in *pm*.

5.4 Performance Evaluation

In this section, we show the performance comparison between using CSR and using CRMF shown in Section 4.2.2, and also show the performance of issuing certificate in the server.

We first tested generating 500 requests in CSR format in the device (Raspberry Pi 4) to emulate the certificate request from the device, and then tested generating 500 requests in CRMF format in *pm* (Intel(R) Core(TM) i7-12800H) considering the server generates the request on behalf of the device.

As depicted in Fig. 11, handling computation of 500 requests in the device and in the server shows approximately 20 times difference. Thus deploying CRMF with 'RAverified' could be considered when the performance is critical requirement, as well as the device has lack of capability to generate CSR by itself.

For the certificate issuing, we tested certificate generation on *km* in macOS on M3 Max Processor. *pm* is sending CSR or CRMF with 'createCertificate' request to *km* over NATS, and *km* issues and sends the certificate back to *pm* over NATS. As depicted in Fig. 12, handling 1 certificate approximately takes 0.18 seconds, 100 certificates takes 9.2 seconds.

Implementer could refer to the time comparison and time estimation in the deployment planning.

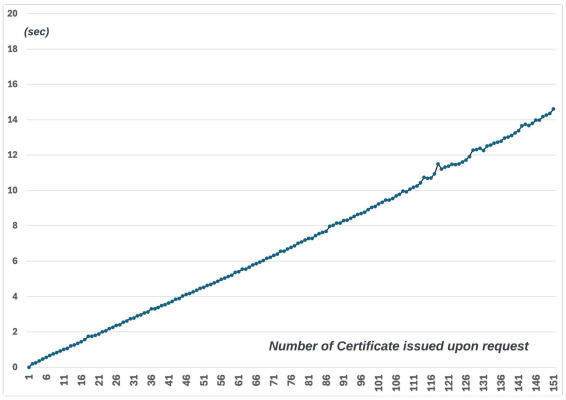

Figure 12: Time that *pm* sends CSR/CRMF and receives the certificate.

6 RELATED WORK

In the study of "Securing IoT Microservices with Certificates" by (Pahl and Donini, 2018) addresses security issues in IoT systems where microservices operate on distributed, resource-limited nodes. The author proposes a certificate-based approach to maintain authentication, accountability, and data integrity throughout the service lifecycle, from development to updates. Another paper proposes a framework for securely managing and rotating secrets in cloud-based microservices (Singh and Aggarwal, 2023). It automates secret rotation, balances centralized and decentralized strategies, and ensures minimal service disruption. The approach includes key encryption, periodic rotation, access control, and automated revocation, enhancing overall system security. The framework focuses on scalability, continuous monitoring, and robust auditing, providing a secure and efficient solution for cloud environments. Finally, (Krämer et al., 2019) addresses securing applications in smart city cloud environments using microservices. It outlines challenges such as data privacy, authentication, and inter-service communication. The authors propose a multi-layered security model that combines role-based access control, encryption, and trusted communication protocols.

Our paper distinguishes itself from existing research by focusing on implementing Public Key Infrastructure (PKI) through a microservice-based architecture tailored for small and medium-sized enterprises (SMEs). Unlike traditional cloud-centric or monolithic PKI systems, your solution prioritizes on-premises scalability, modular updates, and minimal resource overhead, using Docker-based microservices and Commercial Off-The-Shelf (COTS) products. This design avoids the high costs, vendor lock-ins, and complexities of Kubernetes, offering SMEs greater control, security, and adaptability. By focusing on localized infrastructure, your approach ensures scalability, cost efficiency, and robust security without relying on cloud dependencies, making it a practical and accessible choice for businesses with limited budgets and resources.

7 CONCLUSION

This paper presents a cost-efficient security model for device lifecycle management, integrating key provisioning, certificate management, and software management through a microservice architecture. The model supports small-sized organizations and diverse operational environments, ensuring cost efficiency, flexibility, and scalability.

A case study demonstrates the model's implementation using commercially off-the-shelf (COTS) solutions. This approach allows organizations to establish a security infrastructure that meets current requirements and adapts to future challenges and changes in the security landscape, maintaining robust security measures against evolving threats.

REFERENCES

Al Blooshi, S. and Han, K. (2022). A study on employing uptane for secure software update ota in drone environments. In *2022 IEEE international conference on omni-layer intelligent systems (COINS)*, pages 1–6. IEEE.

Chandramouli, R. (2019). Microservices-based application systems. *NIST Special Publication*, 800(204):800–204.

Chia, J., Heng, S.-H., Chin, J.-J., Tan, S.-Y., and Yau, W.-C. (2021). An implementation suite for a hybrid public key infrastructure. *Symmetry*, 13(8):1535.

Dua, A., Barpanda, S. S., Kumar, N., and Tanwar, S. (2020). Trustful: A decentralized public key infrastructure and identity management system. In *2020 IEEE Globecom Workshops (GC Wkshps*. IEEE.

Gaudio, D., Reichel, M., and Hirmer, P. (2020). A life cycle method for device management in dynamic IoT environments. In *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security*. SCITEPRESS - Science and Technology Publications.

Hasselbring, W. (2016). Microservices for scalability: Keynote talk abstract. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 133–134.

Howell, G., Franklin, J. M., Sritapan, V., Souppaya, M., and Scarfone, K. (2023). Guidelines for managing the security of mobile devices in the enterprise. Technical

report, National Institute of Standards and Technology.

Huang, W., Ganjali, A., Kim, B. H., Oh, S., and Lie, D. (2015). The state of public infrastructure-as-a-service cloud security. *ACM Computing Surveys (CSUR)*, 47(4):1–31.

Jain, A., Gupta, S., Vyas, M., Pathy, D., Khare, G., Rajan, A., and Rawat, A. (2018). Open source ejbca public key infrastructure for e-governance enabled software systems in rrcat. In *ICT Based Innovations: Proceedings of CSI 2015*, pages 127–139. Springer.

Kohnhäuser, F., Meier, D., Patzer, F., and Finster, S. (2021). On the security of iiot deployments: An investigation of secure provisioning solutions for opc ua. *IEEE access*, 9:99299–99311.

Krämer, M., Frese, S., and Kuijper, A. (2019). Implementing secure applications in smart city clouds using microservices. *Future Generation Computer Systems*, 99:308–320.

Kuppusamy, T. K., DeLong, L. A., and Cappos, J. (2017). Securing software updates for automotives using uptane. *login Usenix Mag.*, 42(2).

Kwon, S., Jeong, J., and Shon, T. (2018). Toward security enhanced provisioning in industrial iot systems. *Sensors*, 18(12):4372.

Miettinen, M., van Oorschot, P. C., and Sadeghi, A.-R. (2018). Baseline functionality for security and control of commodity IoT devices and domain-controlled device lifecycle management. *arXiv [cs.CR]*.

Oluyede, M. S., Mart, J., Olusola, A., and Olatuja, G. (2024). Container security in cloud environments. *ScienceOpen Preprints*.

Pahl, M.-O. and Donini, L. (2018). Securing iot microservices with certificates. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE.

Pandiya, D. K. (2021). Scalability patterns for microservices architecture. *Educational Administration: Theory and Practice*, 27(3):1178–1183.

Rasyid, I. F., Zagi, L. M., et al. (2022). Digital forensic readiness information system for ejbca digital signature web server. In *2022 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 177–182. IEEE.

Rossi, F., Cardellini, V., and Presti, F. L. (2020). Hierarchical scaling of microservices in kubernetes. In *2020 IEEE international conference on autonomic computing and self-organizing systems (ACSOS)*, pages 28–37. IEEE.

Schaerer, J., Zumbrunn, S., and Braun, T. (2022). Veritaa: A distributed public key infrastructure with signature store. *Int. J. Netw. Manage.*, 32(2).

Singh, A. and Aggarwal, A. (2023). Microservices security secret rotation and management framework for applications within cloud environments: A pragmatic approach. *Journal of AI-Assisted Scientific Discovery*, 3(2):1–16.

Soos, G., Kozma, D., Janky, F. N., and Varga, P. (2018). IoT device lifecycle – a generic model and a use case for cellular mobile networks. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 176–183. IEEE.

Tan, S.-Y., Yau, W.-C., and Lim, B.-H. (2015). An implementation of enhanced public key infrastructure. *Multimed. Tools Appl.*, 74(16):6481–6495.

Ugale, S. and Potgantwar, A. (2023). Container security in cloud environments: A comprehensive analysis and future directions for devsecops. *Engineering Proceedings*, 59(1):57.