DOI: 10.1561/9781638285076.ch9

Chapter 9

# Digital Twins for Secure Software Updates to Maintain IoT Device Trustworthiness

*By Rustem Dautov, Hui Song and Shukun Tokas*

Software updates are critical in maintaining the security and reliability of IoT devices, acting as a key response strategy to address identified vulnerabilities and enhance trust scores. In dynamic IoT environments, devices can become susceptible to new threats, and without timely updates, their trustworthiness diminishes. Ensuring regular and secure software updates is thus essential to protect these devices from exploitation, maintain their functionality, and safeguard the broader network they are part of.

The Trusted Computing Group (TCG) has developed the Guidance on Secure Software Updates in Embedded Systems [1], a reference architecture that outlines the key steps required to manage software updates securely. This includes secure development, signing, distribution, installation, and post-installation verification. While the TCG Guidance provides a solid foundation for ensuring the security of updates, it remains a general framework that does not specify how to implement a fully integrated solution. In practice, the individual steps of the update lifecycle are often addressed separately, resulting in disjointed approaches and gaps in the security and functionality of updates.

Our approach introduces Digital Twins as a powerful solution to enhance the software update lifecycle. Digital Twins provide real-time, virtual representations

of IoT devices, allowing for continuous monitoring of device status and contextual properties. This integration allows the Digital Twin to act as a central point for managing software updates, ensuring timely delivery, guaranteed installation through the Desired-Reported Property pattern, and granular control over which devices receive the updates. Digital Twins also support scalability, enabling efficient management of software updates across large fleets of IoT devices. Additionally, Digital Twins ensure state persistence, allowing devices to return to their desired state following updates, recoveries, or rollbacks.

The expected benefits of this approach are manifold. By leveraging Digital Twins, IoT device updates can be more timely, reliable, and precise, with reduced service disruption and enhanced security. The ability to integrate updates with real-time contextual information ensures that updates are tailored to each device's needs, increasing operational efficiency. This approach represents an adaptable and scalable solution to the challenges of secure software updates in modern IoT ecosystems.

## 9.1 Introduction

In the rapidly advancing domain of connected IoT devices, the significance of secure software updates (SSUs) cannot be overestimated. These devices, ranging from low-level resource-constrained IoT sensors to more capable edge gateways, are integral to many modern industries, including the ones represented by the ERATOSTHENES use cases: Smart Healthcare, Connected Vehicles and Industry 4.0. In all these domains, ensuring the security and reliability of the software that governs these devices is paramount for system functionality. To this end, this chapter outlines the design and implementation of the SSU mechanism in ERATOSTHENES, highlighting its critical role in maintaining the integrity and efficacy of IoT devices, drawing upon the principles established in the 'Guidance for Secure Update of Software and Firmware on Embedded Systems' by the Trusted Computing Group (TCG).

### 9.1.1 Reference Architecture: Secure Software Update Lifecycle

The SSU lifecycle consists of five essential steps designed to ensure that software updates are applied safely, maintaining the security and functionality of the system. These steps are typically referenced from the TCG Guidance, which provides a reference architecture rather than prescriptive instructions on how to implement specific solutions. This flexibility allows developers to adapt the principles to their needs, but it often results in disjointed implementations of the lifecycle steps, rather

than a unified, holistic approach. Schematically depicted in Figure 9.1, the SSU lifecycle includes the following steps:

1. **Secure Development:** This step focuses on the secure creation of the software update itself. It involves following security best practices during the design and development stages to ensure that the software is free of vulnerabilities and prepared for safe distribution.
2. **Secure Signing:** Once developed, the software update is cryptographically signed to guarantee its authenticity and integrity. This step ensures that the update comes from a trusted source and has not been altered or tampered with before distribution.
3. **Robust Distribution:** The signed update is securely distributed to devices. This involves ensuring that the update is delivered to the correct devices in a reliable manner, preventing unauthorised interception or modification during transmission.
4. **Secure Installation:** After distribution, the update is securely installed on the device. This step ensures that the update is applied without errors or security risks, maintaining system stability and trust.
5. **Post-installation Verification:** Following installation, this step ensures that the update was successfully applied and that the system is functioning correctly. It verifies the integrity of the new software and checks that the device's trust and security parameters remain intact.

While the TCG Guidance outlines these steps as key elements of a secure update lifecycle, it does not dictate how to build or implement these processes. As a result, in practice, organisations often implement individual steps in isolation, without integrating them into a cohesive, end-to-end architecture. This fragmented approach can weaken the overall security posture, as the lack of coordination between steps may introduce vulnerabilities or gaps in the update process.

### 9.1.2 Proposed Approach at a Glance: Why Digital Twins?

Digital Twins (DTs) can significantly enhance the SSU lifecycle by acting as a centralised, real-time reference for the status of installed software across IoT devices. By mirroring the physical device in a virtual environment, DTs provide a comprehensive view of each device's software, making the update process more efficient and reliable. When combined with other tools, DTs enable a streamlined approach to managing updates, improving the delivery, control, and verification of software changes. These practical benefits are summarised as follows:

- **Timeliness** is a key benefit of using DTs. By continuously collecting real-time contextual information from IoT devices, DTs ensure that software

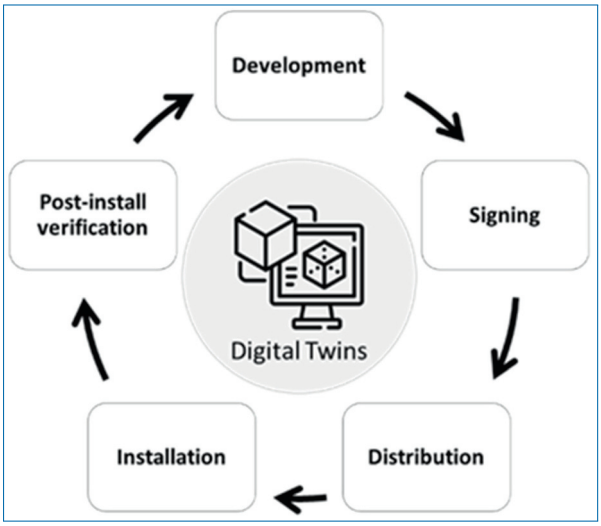


Figure 9.1. Secure software lifecycle supported by the digital twins.

updates are delivered promptly. This real-time insight allows the system to push updates with minimal service disruption and downtime, as updates are deployed when conditions are optimal for each device, such as when it is idle or less critical to operations.

- **Guaranteed delivery** is facilitated by the use of the Desired-Reported Property pattern. In this approach, the desired state (new software version) is sent to the DT, while the device reports back its current state. Even if a device is temporarily offline, the DT keeps track of the desired state. As soon as the device comes back online, the system ensures the update is installed, guaranteeing no missed updates regardless of device availability.
- **Granular control** over the update process is another advantage. DTs allow for precise management of updates, whether targeting a single device, specific groups, or an entire fleet. The cyber-physical-social context of each device can be taken into account, ensuring that updates are deployed only when suitable. For example, more critical devices can receive updates first, or updates can be staggered based on geographical location, device type, or operational role.
- **Scalability** is greatly enhanced by DTs. They enable the management of large fleets of devices, potentially consisting of hundreds or even thousands of devices. By providing a digital representation of each device, updates can be deployed and monitored across vast networks, ensuring all devices receive the correct updates while keeping track of their individual statuses.
- **A sandbox environment** is another significant benefit. DTs provide the capability to test updates in a virtual space before pushing them to the physical

devices. This allows organisations to validate updates in a safe, isolated environment, identifying potential issues before they affect the real-world devices. This mitigates risks and ensures that updates are fully functional before deployment.

- **State persistence for upgrades and recovery** is another critical feature of DTs. By keeping the history of the twin updates, the system can restore a device to its desired state following an upgrade or recovery process. This ensures that if an update fails or a device requires a recovery, it can seamlessly return to its correct configuration, maintaining the continuity of operations.

These benefits will be further revisited in this section with a more detailed explanation of how the proposed DT-based approach can implement them. Noteworthy, this approach is by no means a complete, standalone solution for the SSU lifecycle. Rather, it serves as an end-to-end point of integration, where various existing tools and approaches that address individual steps of the SSU lifecycle can converge. DTs act as a centralised framework that ties together different components, such as secure signing tools, robust distribution systems, and post-installation verification mechanisms. While it provides a unified view and control of the update process, this approach can and should be further extended and complemented with additional tools. For example, enhanced security mechanisms, automated verification systems, and advanced simulation tools can be incorporated to strengthen specific stages of the lifecycle, ensuring a more robust and holistic update management system.

### 9.1.3 Positioning within ERATOSTHENES

ERATOSTHENES's vision is to build a robust and secure ecosystem for devices, ensuring that each device operates reliably and securely throughout its lifecycle. This ecosystem aims to safeguard application data, ensure the uninterrupted provisioning of business services, and maintain compliance with applicable regulatory requirements. SSUs are essential for implementing of this vision, enabling the seamless and safe evolution of device capabilities while protecting against emerging cyber-threats. In this context, SSUs ensure that vulnerabilities are patched promptly, reducing the risk of malicious interference or malfunction, thus aligning with the TCG's emphasis on maintaining device integrity through timely and trustworthy updates.

In the context of ERATOSTHENES, a core pillar is the use of SSUs as a reaction strategy to an event that reduces the trust level of an IoT device. Thus, the SSU mechanism is one of the possible instruments that ERATOSTHENES users can leverage in order to bring the IoT devices back into a trustworthy state.

Therefore, the scope of this research effort encompasses the development and implementation of a comprehensive SSU framework across all IoT devices within

the ERATOSTHENES network. This includes conducting thorough security assessments of existing devices, integrating secure update mechanisms, and establishing an automated and secure software delivery system. More specifically, the SSU process can be triggered after a failed trust assessment, i.e., when the actual calculated trust score of an IoT device is lower than the expected score. This essentially signifies the need to identify potential vulnerabilities and apply mitigation measures.

## 9.2 Conceptual Architecture of the Secure Software Update Mechanism

We now proceed with the design of the SSU functionality, as depicted in Figure 9.2. In this architecture, we distinguish between stakeholders and functional elements (i.e., software components).

### 9.2.1 Main Stakeholders

The main stakeholders of this architecture are the following:

- **Device Manufacturer** plays a crucial role in the overall ERATOSTHENES ecosystem, not only by producing the IoT hardware but also by developing the associated software. The expected version of the software stack to be deployed for each device model is specified as part of the MUD profile [9]. The device manufacturer may either develop the software themselves or subcontract trusted third parties (**Software Developer**) to ensure specialised expertise and security. Regardless of the chosen development approach, the integrity and security of the software updates remain paramount. The manufacturer (or a trusted third party acting on its behalf) sign the software updates with a private key, ensuring that each update can further be authenticated and verified. The corresponding public key is embedded in the manufactured IoT devices, enabling the high-end devices to verify the authenticity of updates upon receipt.[1] This cryptographic method ensures that only verified, untampered updates are installed, maintaining the security and reliability of the device.
- **System administrator** in the context of ERATOSTHENES possesses in-depth knowledge of the application domain and its specific security-related requirements and is thus responsible for providing information on

1. Please note that this approach primarily applies to relatively capable IoT devices that have the sufficient computing capabilities for asymmetric cryptography.

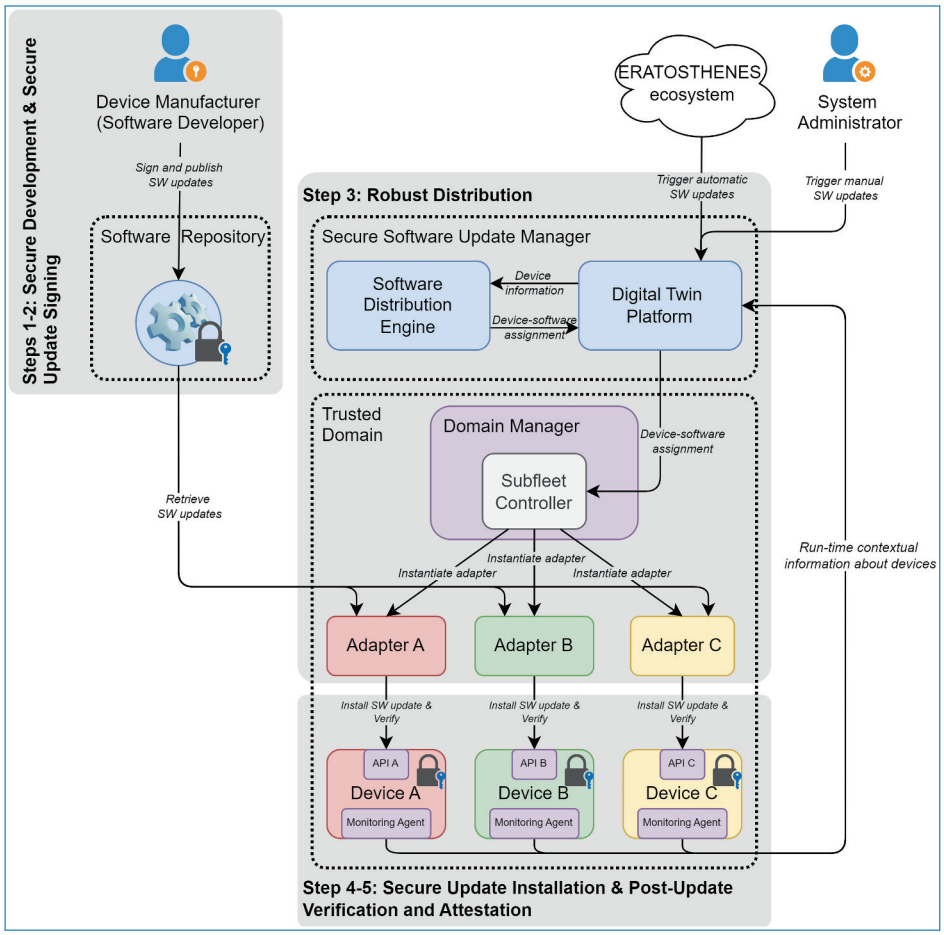


**Figure 9.2.** Conceptual architecture of the secure software update mechanism.

domain-specific threats and vulnerabilities for risk assessment. In addition to these responsibilities, the system administrator also oversees the SSU process and ensures that software patches are applied correctly and securely across all devices. Using the DT Platform, the system administrator can manually trigger updates if needed, to provide timely response to emerging threats or vulnerabilities.

### 9.2.2 Main Functional Elements

The main functional elements of the architecture are the following:

- **ERATOSTHENES ecosystem** collectively represents the rest of the ERATOSTHENES components which may trigger the SSU process. More

specifically, the main reason for automatic software updates envisioned by ERATOSTHENES is a reduced trust score of an IoT device caused by identified vulnerability in the existing software. The calculated trust score being lower than expected signifies the need to identify potential vulnerabilities and apply necessary mitigation measures. Noteworthy, while the triggering of updates can be fully automatic, in ERATOSTHENES we assume that the SSU process may also rely on the involvement of the system administrator who oversees this whole process.

- **Digital Twin Platform** maintains a live view of the managed device fleet, i.e., a collection of continuously updated DTs for each managed IoT device. The DT Platform is the central element of the whole SSU functionality, as it maintains an up-to-date representation of managed devices. This enables secure and timely software updates by providing a comprehensive view of each device's configuration and operational state. For the automatic triggering of software updates, the DT Platform is equipped with APIs (e.g., HTTP entry-points and MQTT listeners), so that the rest of the ERATOSTHENES components can interact with it. As already outlined, the use of DTs introduces the following benefits: timeliness, guaranteed delivery, granular control, scalability, a sandbox environment, and state persistence.
- **Distribution Engine** takes as input the current contextual information about managed IoT devices – on the one hand, and software assignment constraints – on the other. Based on its internal knowledge and rules, it then assigns a specific software version to a specific target device, i.e., generates device-software assignment pairs as an output. Targeted, context-aware assignment of software updates in the IoT using DTs is essential for ensuring that each device receives updates tailored to its specific configuration and operational environment. This approach minimises the risk of incompatibility and maximises the effectiveness of updates.
- **Subfleet Controller** is deployed and runs within the ERATOSTHENES trusted domain (i.e., close to devices in the secure network domain) and is responsible for interaction with the DT Platform on behalf of a subfleet of devices (e.g., a subfleet of connected healthcare gateways in remote patient monitoring). It receives software update commands from the central DT Platform through its northbound interface and propagates them to the downstream IoT devices via its southbound interface using device- or platform-specific Adapters. This setup ensures efficient and secure dissemination of updates, tailored to the specific needs of local devices, while maintaining synchronisation with the central management system.
- **Software Repository** serves as a central distribution point of SSUs for devices. This repository can be a private server managed by the manufacturer

or a public cloud-based service, such as Docker Hub,[2] which is widely used for distributing containerised applications. Acting as a central hub, the repository ensures that updates are securely stored and can be accessed programmatically by the adapters and/or devices. Secure protocols like HTTPS are employed to protect the integrity of the data in transit, and access control mechanisms are in place to restrict update uploads and modifications to authorised personnel only.

- **Adapters** are device- or platform-specific software components responsible for interacting with devices and enacting the actual software updates on devices. They are instantiated and run along-side the subfleet controller in the trusted domain and depending on the type of devices in its sub-fleet, the subfleet controller will instantiate a corresponding adapter. These adapters are tailored to the unique characteristics and protocols of each device type or platform within the IoT fleet. By interfacing directly with the devices through standardised or proprietary APIs, they ensure that updates are deployed efficiently and securely.
- **Devices** are diverse IoT assets that ERATOSTHENES already deals with (e.g., healthcare gateways, smart vehicle onboard units, and smart industrial appliances), as well as a wider range of network-connected IoT devices with similar characterises. In ERATOSTHENES, we primarily focus on relatively capable devices equipped with sufficient hardware capabilities (e.g., network bandwidth, CPU, storage). Devices expose some APIs specific to their hardware platform, network interfaces, OS, software stack, etc., which are used by the Adapters to enact the software update process.
- **Monitoring Agents** run on the devices and serve to collect real-time contextual information. These agents gather data on device status, operational parameters, environmental conditions, and other relevant metrics. This information is then transmitted back to the central DT Platform where it is represented as reported properties in DTs. By continuously updating these properties, the platform maintains an accurate and dynamic DT model of each device, facilitating informed decision-making for the following SSU activities, as well as for stateful recovery.

### 9.2.3 Triggering Software Updates with the Desired-Reported Property Pattern

The **Desired-Reported Property** pattern in DTs is a key mechanism for managing and synchronising the state of physical assets and their virtual counterparts, as

2. https://hub.docker.com/

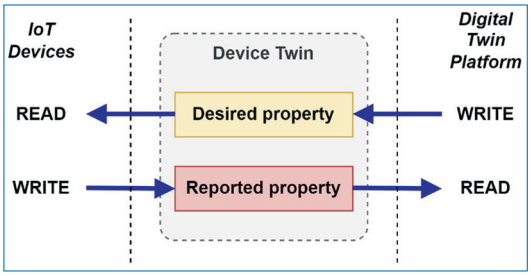


**Figure 9.3.** Desired-reported property pattern in digital twins.

depicted in Figure 9.3. This pattern involves two primary elements:

- **Desired Properties:** These are the target states or configurations that the System Administrator wants the physical asset to achieve. They are set within the DT and serve as directives for how the asset should be configured or operated. For instance, in the context of SSUs, the desired property might be the target software version to be deployed.
- **Reported Properties:** These reflect the current state or configuration of the physical asset as detected and communicated by sensors or monitoring systems. The reported properties are continuously updated to provide a runtime status of the asset's condition. Continuing with the SSU example, the reported property would be the current software version that the device is running. Reported properties, and especially evidence on the correct device configuration (which entailed evidence on the device's correct software version running), is achieved through the trusted computing enablers of ERATOSTHENES.

The interplay between these two properties allows for effective monitoring and control. When a discrepancy is detected between the desired properties and the reported properties, the DT Platform can trigger actions to align the physical asset with the desired state. This might involve sending commands to the asset to adjust its configuration (e.g., upgrade the software version), or it might involve alerting operators to take corrective actions. This pattern is particularly valuable in the ERATOSTHENES use case scenarios requiring precise control and automation, such as telemedicine using remote patient monitoring, smart manufacturing, and intelligent vehicle services. It ensures that the DT remains an accurate and actionable representation of the physical asset, enabling proactive maintenance, optimisation, and fault detection.

### 9.2.4 Context-Aware Software Assignment

Software assignment using DTs in devices leverages detailed digital representations of physical devices, incorporating multi-dimensional contextual properties, to

assign software updates efficiently. More specifically, the cyber-physical-social context of monitored devices encompasses several dimensions [2]. The cyber aspects include the traditional hardware and software properties of the devices. Physical aspects refer to environmental conditions such as the device's location, surrounding temperature, and time of day, which can also affect device performance and availability. Social aspects involve the actual human user, considering factors like specific medical conditions, usage patterns, and service subscription types, all of which influence how the device is used and monitored.

Target conditions for software updates can be defined in two primary ways, each offering different levels of control and flexibility. One option is for the **Device Manufacturer** (**Software Developer**) to specify the conditions, such as identifying the security vulnerability the update addresses and listing the specific device models that are affected and require the patch. This ensures that the update is applied where it is most needed, addressing known issues in the devices it was designed to protect. Another, more flexible approach is for the System Administrator to manually define the target conditions. By leveraging their knowledge and experience, the Administrator can take a more fine-grained approach, tailoring the update deployment based on the contextual properties of each device. These properties, collected and maintained by the DT Platform, allow the Administrator to consider factors such as device location, usage patterns, and current operational state. This method enables highly targeted updates, ensuring that the right devices are updated at the right time, maximising both security and operational efficiency.

In both cases, the distribution engine uses these context-aware DT representations to match target software requirements with the appropriate devices. This can be implemented in three possible scenarios:

- **One-to-One Scenario:** the distribution engine assigns a specific software update to a single device based on its unique contextual properties. For example, if a DT of a particular device indicates that it requires a firmware update to fix a known vulnerability, the Distribution Engine will push the update directly to that specific device using its unique ID as the target condition. The Desired-Reported Property pattern is used to confirm the device reaches the desired state after the update, ensuring the device operates securely and effectively after the installation.
- **One-to-Many Scenario:** the Distribution Engine assigns software updates to a group of devices sharing certain contextual properties. For instance, a hospital might have several patient monitoring systems that require an update to enhance data encryption protocols. The DTs of these monitors, which may include similar models or devices within the same department, will indicate they all need the update. The Distribution Engine identifies

these commonalities and assigns the update to all relevant devices, ensuring consistency and efficiency. So if a vulnerability is identified in more than one device, a patch can be applied to all of them at once. This approach facilitates the update process across multiple devices, leveraging batch processing capabilities to maintain high standards of data security.

- **One-to-All Scenario:** the Distribution Engine broadcasts a software update to all devices within the network. This is typically done for critical updates, such as a security patch addressing a widespread vulnerability. For example, if a new regulation requires enhanced cyber-security measures across all IoT devices, the DTs provide the necessary contextual properties to ensure each device is eligible and capable of receiving the update. For example, if a vulnerability is identified for a single device of a specific model, a patch can be applied to all other devices of this type in a scalable and timely manner. The Distribution Engine then ensures that this update is deployed across the entire fleet of devices, leveraging the DTs to monitor and validate the update process.

In each of these scenarios, the Desired-Reported Property pattern is used to confirm the devices reach the desired state after the update, to make sure they operate securely and effectively after the installation.

## 9.3 Implementing the Software Management Lifecycle

The described conceptual architecture for SSUs in ERATOSTHENES is aligned with the already mentioned TCG Guidance. The software update lifecycle is part of the runtime phase of the ERATOSTHENES architecture. It is expected that the secure device domain is already established, with the IoT devices enrolled and the domain manager up and running.

The SSU lifecycle begins with the development step, which employs secure coding practices. Updates are cryptographically signed to verify their authenticity and integrity, ensuring that only verified updates can be installed on the device. The distribution step ensures updates are assigned and delivered securely, using encrypted communication channels and secure servers to prevent interception or tampering. In the deployment phase, devices must authenticate the updates before installation. This involves verifying the cryptographic signatures to confirm the update is from a trusted source and has not been altered. Post-deployment, the verification and attestation step involves validating the update's integrity and functionality, monitoring for any issues or anomalies that might arise. All these lifecycle steps are also included in Figure 9.2 as grey boxes in the background. In the following subsections, we will look into each of these steps in more detail, describing how the proposed ERATOSTHENES SSU functionality implements them.

### 9.3.1 Step 1: Secure Development of Software Updates

In ERATOSTHENES, developing software patches can take place in response to reduced trustworthiness levels of devices. Thus, it is seen as part of the response mechanism to identified vulnerabilities. Implementing secure development for devices goes beyond the official scope of ERATOSTHENES, thus, we assume that developers and service providers follow best practices for secure software development. Essential practices include: incorporating secure coding practices throughout the software development lifecycle to avoid vulnerabilities like buffer overflows and SQL injection; regular code reviews and the use of static analysis tools are essential for early detection of vulnerabilities; conducting extensive security testing, including penetration testing, to identify and mitigate potential issues before deployment; employing robust cryptographic measures to protect data, and ensure that software updates are signed and verified to maintain integrity [9, 11].

### 9.3.2 Step 2: Secure Signing of Software Updates

In the first place, implementing secure update signing for IoT devices (depicted in Figure 9.4 as a simplified sequence diagram) requires robust key management and thorough signing of software updates. In ERATOSTHENES, we assume that Device Manufacturers generate and manage a pair of cryptographic keys: a private key for signing updates and a corresponding public key for verifying them. The private key is stored securely, while the public key is embedded within the IoT devices during the manufacturing process, enabling these devices to verify the authenticity of received updates. For the devices capable of symmetric cryptography, it can be the case that a symmetric SSU key is established in each device during the manufacturing process.

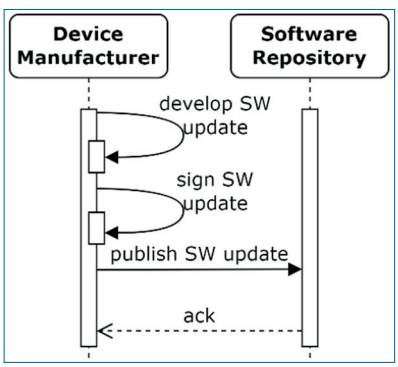


**Figure 9.4.** Steps 1–2: Secure development and signing of software updates.

When a software update, such as a Docker container or a binary executable file, is prepared for deployment following the secure development phase, the manufacturer uses their private key to create a digital signature. This process involves generating a hash of the update file and encrypting this hash with the private key. The resulting digital signature is then attached to the software update package. For updates involving Docker containers, the process is similar. The entire container image is signed by hashing its content and creating a digital signature. Devices that run Docker can verify the container signature before deploying it, ensuring that the container has not been altered. After generating a digital signature using the private key, the signed software updates are securely uploaded to the designated software repository, which can be a private server managed by the Device Manufacturer or a public cloud-based service. At this point, the signed software update, coupled with its signature, is ready for secure distribution to the devices.

### 9.3.3 Step 3: Robust Distribution

To implement robust distribution of software updates for IoT devices (Figure 9.5), leveraging DTs and the Desired-Reported Property pattern is essential. As previously explained, DTs provide real-time virtual representations of each IoT device, capturing their current status, including operating system, software version, execution traces, and other critical data. The Distribution Engine uses the detailed information from DTs to determine which devices need updates [3]. By comparing the desired state (i.e., the latest software version) with the reported state (i.e., the current software version on the device), the engine can identify discrepancies and target the necessary updates.

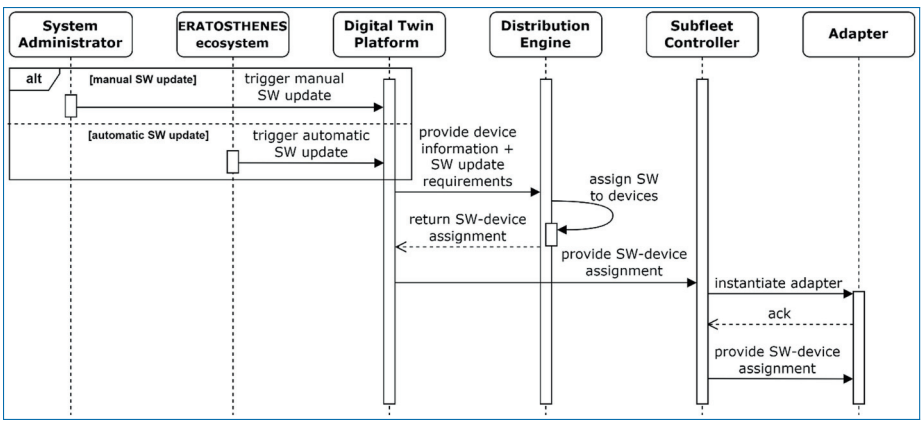


**Figure 9.5.** Step 3: Robust distribution of software updates.

For a single device (i.e., one-to-one scenario), the Distribution Engine checks the DT's reported properties and pushes the update if the device is not up to date. For a subset of devices (i.e., one-to-many scenario), the engine filters devices based on specific criteria, such as those running an outdated version or those with a particular operating system. It then pushes the update to this group, ensuring all selected devices meet the desired state. For fleet-wide updates (i.e., one-to-all scenario), the engine initiates a fan-out distribution to all devices. This ensures uniformity across the entire network, with each device's DT confirming receipt and installation of the update, maintaining consistent software versions across the fleet.

The distribution itself takes place over secure channels, ensuring data integrity and preventing unauthorised access. The process includes continuous monitoring of the update status, using the DTs to track progress and verify successful installations.

### 9.3.4 Step 4: Secure Update Installation

Secure installation of software updates on IoT devices (Figure 9.6) requires addressing the diversity in hardware architectures, operating systems, execution environments, network interfaces, and APIs. This implementation relies on specific software Adapters developed for each device type, ensuring a seamless installation process tailored to the unique requirements of each device [3, 4].

In addition to the Adapters, the Subfleet Manager is another important software component installed within the same secure domain as the devices. It coordinates and manages the update process across a diverse range of IoT devices, ensuring that each device receives the appropriate updates securely and efficiently. The Subfleet Manager acts as intermediary between the central SSU server (i.e., DT Platform)

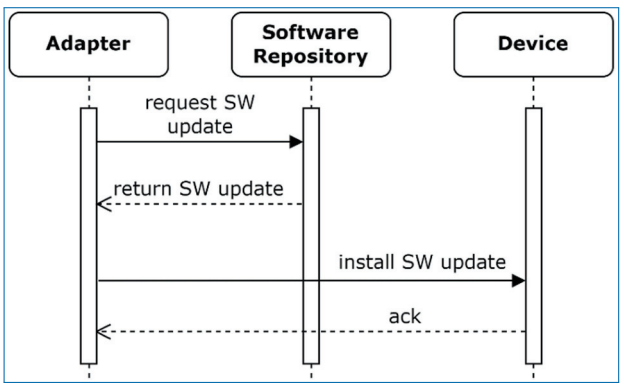


**Figure 9.6.** Step 4: Secure installation of software updates.

and the IoT devices; its main task is to receive software update commands from the DT Platform on the northbound interface and to route them to the assigned device on its southbound interface [5]. Upon receiving a command, the Subfleet Manager identifies the specific Adapter required for the target device type and instantiates it accordingly. It then forwards the software update command to the designated Adapter, which then enacts the correct and secure execution of the update process.

The installation process varies depending on the device type but generally involves securely transferring the update package to the device and initiating the installation process via the corresponding API. The Adapter handles specific commands and procedures required for installation on different operating systems and execution environments. It monitors the installation process in real-time to ensure it completes successfully without interruptions or errors.

Before installation, the Adapter may perform signature verification by fetching the update package from the secure repository and using the device's embedded public key to verify the attached digital signature. This ensures that the update is from a trusted source and has not been tampered with. If verification fails, the installation is aborted, and the issue is logged.

### 9.3.5 Step 5: Post-Update Verification and Attestation

This is the last step in the SSU cycle (Figure 9.7). Local attestation involves verifying the integrity of software updates directly on the device. Each device is equipped with a public key from the manufacturer. These keys are essential for verifying the authenticity and integrity of installed software updates, which are signed with the manufacturer's private key. Before a software update is installed, the device uses this public key to verify the digital signature of the update. This verification ensures that the update has not been tampered with and is indeed from a trusted source.

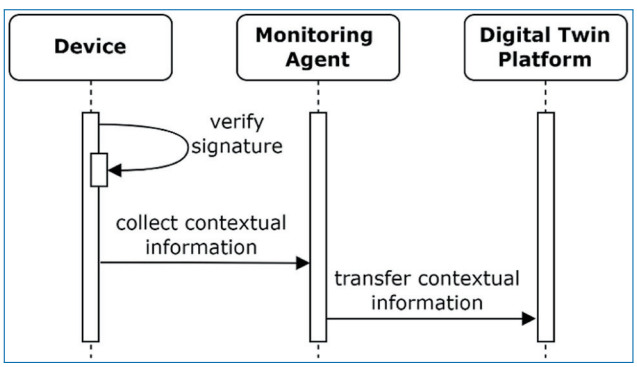


**Figure 9.7.** Step 5: Post-update verification and attestation.

This process can be automated and performed periodically, ensuring the software's continuous integrity.

To further enhance security, continuous monitoring is implemented using device-side Monitoring Agents. These agents run on each device, collecting runtime information such as current software versions, execution traces, and other relevant metrics. This data is sent to the DT Platform at regular intervals. The platform uses this information to maintain an up-to-date virtual representation of each device, enabling real-time monitoring and detection of anomalies or discrepancies.

The continuous monitoring system allows for proactive security measures, as any deviations from expected behaviour can be quickly identified and addressed. By leveraging local attestation supported by continuous monitoring, a robust framework is established to ensure the security and integrity of software updates in devices, thereby safeguarding user data and device functionality.

## 9.4 Proof of Concept

Following the described architecture, the proof-of-concept implementation focuses on adoptability and integrability. Therefore, we have taken an architectural design approach to improve aspects of coupling to application-specific details and separation of concerns. Given the cyber-physical nature of managed IoT devices within the distributed fleet, three main design decisions underpin the design and implementation of the SSU mechanism:

- **Decoupled architecture:** Managed IoT devices may unexpectedly crash, unpredictably lose network connection, and then reappear online. To accommodate such ad-hoc behaviour and prevent bottlenecks, communication between software components is implemented using pub-sub messaging based on MQTT.
- **Asynchronous communication:** For the same reason, it is important that any software updates initiated are guaranteed to be delivered to target devices regardless of how long they stay offline. This also means that the status of each device is continuously monitored and stored to apply the required changes when it gets back online. This can be implemented using the described Desired-Reported Property pattern, which enables real-time monitoring of devices and comparison of reported properties against desired ones.
- **Hierarchical architecture and extensibility:** To prevent single points of failure and distribute the workload within the managed fleet of heterogeneous

devices, devices with similar interfaces are grouped together [3]. This introduces an extra layer of filtering and load-balancing functionality between the centralised SSU engine and numerous devices within the managed fleet. Thanks to this hierarchical architecture, where generic functionality is separated from device-specific implementations, the SSU engine will be able to accommodate new types of devices in the future.

In addition to these main requirements, the design and implementation of the SSU mechanism were driven by several non-functional requirements, such as the availability of a graphical user interface (GUI), user friendliness, and the intention to re-use existing software libraries.

To implement the conceptual architecture of the SSU functionality depicted in Figure 9.2, several software components were developed. These elements correspond to the main functional elements of the conceptual architecture, which we describe in more detail in the following subsections.

### 9.4.1 Digital Twin Platform: Eclipse Ditto

As the technological baseline on top of which we developed the SSU functionality, we used Eclipse Ditto[3] – an open-source framework for building DTs of Internet-connected devices with extensible modelling and built-in querying languages.

As already explained, by using a device management platform, edge application [12] providers can remotely provision, monitor, and maintain their devices, as well as push software updates either manually or in an automated manner whenever a new version is released. In recent years, all these tasks have been underpinned by the prominent concept of DTs. Simply put, DTs are virtual models designed to accurately reflect physical objects equipped with various sensors related to certain aspects of their functionality. They enable the creation of rich digital models of anything physical or logical, from simple assets or products to complex cyber-physical environments. The collected sensor data is relayed to a processing system and applied to the DT.

Furthermore, Ditto acts as middleware, providing an abstraction layer for IoT solutions interacting with physical devices via the DT pattern. It can be seen as a toolkit, providing core functionality (e.g., meta-model, database, different messaging protocols and connectors, REST APIs, etc.), while some other features must be developed by users on top of it (e.g., domain-specific DT models, graphical user interfaces, device-side monitoring agents, etc.). Being part of a larger open-source ecosystem, it can be relatively easily integrated with other technologies from

3. https://eclipse.dev/ditto/

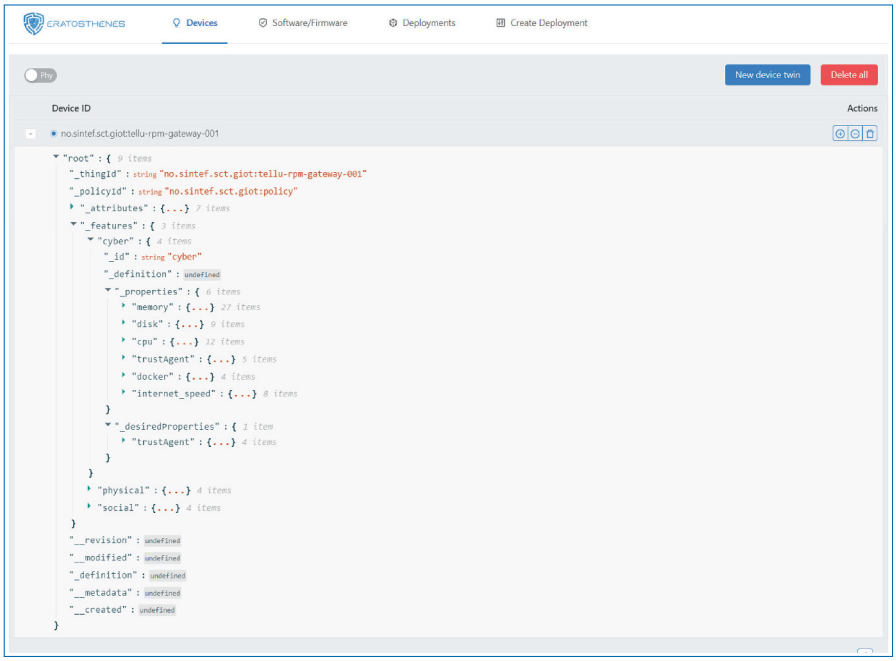


**Figure 9.8.** Front-end interface of the DT Platform for secure software updates.

the Eclipse stack, including various communication protocols, pub-sub messaging, and load balancing. Some of these extensibility capabilities are demonstrated in Figure 9.8, which includes the GUI of the extended DT Platform in the context of ERATOSTHENES. More specifically, the screenshot includes the basic SSU functionality accessible to the System Administrator via a DT representation in the context of the Smart Healthcare use case on connected healthcare gateways.

### 9.4.2 Software Assignment Engine: Ditto Meta-Model + Resource Query Language

Ditto offers developers an extensible meta-model [2] (depicted in Figure 9.9) that, in its simplest form, enables modelling physical entities, referring to them as **Things**, using a JSON schema with the following key concepts:

- **Thing** is the top-level modelling concept for describing physical assets.
- **Definition** is included in every **Thing** (and optionally in **Features**) and essentially represents a URI linking to an external Web of Things (WoT) model.[4]

4. https://www.w3.org/TR/wot-thing-description/

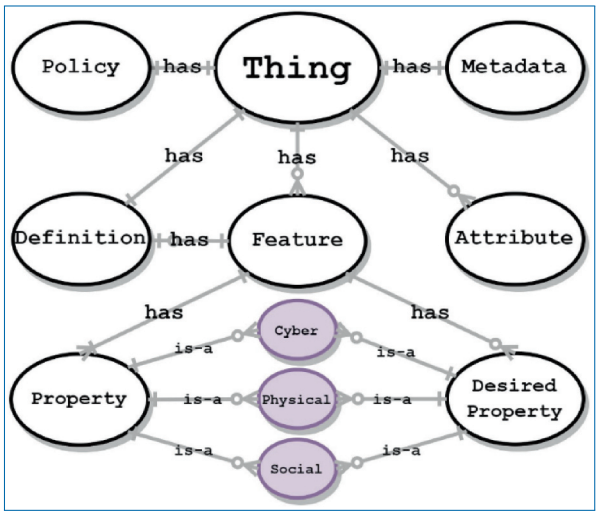


**Figure 9.9.** Digital twin meta-model capturing the multi-dimensional device context (an extension to Eclipse Ditto) [2].

It describes how a **Thing** is structured and what behaviour/capabilities can be expected in an interoperable and standard manner.

- **Policy** enables fine-grained access control configuration. A specific policy defines who and how can access a specific resource.
- **Metadata** is Ditto's internal field to store technical information, e.g., version or creation/modification timestamps.
- **Attributes** are used to model rather static properties of a **Thing**, i.e., values that do not change as frequently as **Features**. They can be of any type and can be used to search for **Things**.
- **Features** are the central modelling concept to capture all run-time data and functionality of a **Thing** in a given application system. Users are allowed to define their own **Features** or extend existing WoT definitions. This is a key enabler for modelling the multi-dimensional context through more fine-grained **Properties**.
- **Properties** are used within **Features** to model individual run-time indicators of a **Thing**, e.g., to manage the status, the configuration, or any fault information. Each **Property** can be either a simple scalar value or a complex JSON object. By using **Properties**, it is possible to implement the prominent Desired-Reported Pattern widely adopted in DTs, wherein sensor measurements are reported upstream while desired configuration updates are pushed downstream until eventually **Properties** and **DesiredProperties** are in sync. As explained, this is the main mechanism for monitoring the state of the deployed software and managing the deployment.

A simplified example of a DT definition (i.e., a connected medical gateway) used by the SSU mechanism is depicted in Listing 9.1. The DT includes all main elements, including the multi-dimensional features. Note the **properties** and **desired_properties** fields, which are used for twin synchronisation and trigger the update procedure of the outdated **software_component_01**.

```
"thingId": " no.sintef.sct.giot:tellu–rpm–gateway–001",
 "policyId": " no.sintef.sct.giot:policy",
 "attributes": {
        "type": "physical_device"
        "manufacturer": "RPM Inc.",
        "cpu_model": "Broadcom BCM2711"
        "platform": " linux/arm/v8",
        "ip_address": "192.168.32.5",
        "os": {
             "name": "Debian",
             "description": " Debian GNU/Linux 11 (bullseye)",
             "version": "11"
        }
 },
 "features": {
       "cyber": {
            "properties": {
                 "trustAgent": {
                      "container_image": "trust–agent"
                      "container_status": "running",
                      "container_version": "0.1"
                 },
                 "docker": {
                      "server_version": "27.2.1"
                 }
            },
            "desired_properties": {
                 " trustAgent": {
                      "container_image": "trust–agent"
                      "container_status": "running",
                      "container_version": "0.2"
                 }
            }
       },
       "physical": {...},
       "social": {...}
       }
 }
```

**Listing 9.1.** A simplified example of a personal healthcare gateway DT.

Eclipse Ditto also offers its users the Resource Query Language (RQL) to search for and manage these DTs efficiently. Using RQL's logical operators, we can perform precise assignments of software updates to IoT/Edge devices based on their DT data. One-to-one assignment involves updating a specific device by querying its unique ID. For instance, if a critical update is needed for a particular device identified by its ID, RQL allows us to target and update just that device:

```
eq(thingId,"no.sintef.sct.giot:tellu-rpm-gateway-001")
```

**Listing 9.2.** One-to-one assignment.

One-to-many assignment leverages RQL to identify and update multiple devices that meet certain contextual requirements. This might include updating devices running an outdated software version, devices running on a specific OS, devices deployed in a particular location, or devices used by patients with a premium subscription:

```
and(eq(features/cyber/properties/trustAgent/container_image,"trust-agent"),
    in(features/cyber/properties/trustAgent/container_version,"0.1","0.2"))
eq(attributes/os/version,"11")
eq(features/physical/properties/location,"hospital")
eq(features/social/properties/subscription,"premium"))
```

**Listing 9.3.** One-to-many assignment.

One-to-all assignment involves updating an entire class of devices. By querying devices in the fleet, we can efficiently distribute the update to ensure all relevant devices are up to date:

```
eq(attributes/type,"physical_device")
```

**Listing 9.4.** One-to-all assignment.

This way, Eclipse Ditto's meta-model and RQL enable accurate and efficient assignment of software updates, enhancing the reliability and security of devices.

#### 9.4.2.1 Subfleet Controller and Adapters

We now discuss the actual installation of software updates onto the managed devices. The design emphasises generality and extensibility, ensuring compatibility with diverse device types and allowing effortless introduction of new device categories. Currently, the system supports three different types of high-end devices (all based on Smart Healthcare use case on remote patient monitoring):

- **Docker:** This type involves any device equipped with a Docker Engine, hosting a software component as a Docker container.
- **Linux SSH:** This category encompasses devices operating on a Linux operating system with open SSH access, wherein the software components operate as basic shell scripts.
- **Device-specific HTTP API (Axis Camera):** Specifically for Axis cameras,[5] the software components run as apps within the camera, utilising the camera's REST API for deployment and lifecycle management.

5. https://www.axis.com/products/network-cameras.

For each of these three, there is an **Adapter**, which oversees the device, handling connection, status checking, basic device info collection related to SSUs, and the actual installation of the updates. Once active, all the Adapters will subscribe to any possible downstream commands from the DT Platform. Whenever there is a software update that requires actions on one of the devices, the Adapter will launch the action on the corresponding device through the device-specific APIs.

We use the Docker Adapter as an example [8]. For devices that support a Docker environment, a software component operates as a Docker container. The essential requirement is for the device to enable the Docker Engine, which provides a REST API for communication. This Adapter operates on the Domain Manager in the same local network with the target device. It utilises the REST API to remotely communicate with the Docker Engine on the device. The device DT holds key information such as the device's IP and the Docker Engine API port. This information allows the Adapter to call the info method of the Docker Engine to retrieve additional metadata about the device, including CPU architecture, OS distribution, and Docker version. This method also serves as a ping operation, ensuring the device is connected.

For installing a software update, the Adapter performs several sequential steps. It begins by downloading the Docker image using the URL provided in the device twin. Following this, it invokes the **loadImage** method of the Docker Engine to upload the image package into the device. To start the software, the Adapter calls **runContainer**, launching a Docker container from the loaded image. Given that loading an image and initialising a container may be time-consuming, the Adapter maintains continuous communication with the Docker Engine.

#### 9.4.2.2 Device-side Monitoring Agents

As previously explained, Monitoring Agents are deployed on devices to collect contextual information, which is then sent to the DT Platform. This information includes hardware and software status, environmental conditions, and user interactions, creating a comprehensive overview of each device's state and performance. By providing real-time data, Monitoring Agents close the monitoring loop, enabling continuous oversight and proactive management of the device fleet.

One of the key functionalities of the Monitoring Agents is to track the successful installation and execution of software updates. They verify that updates have been applied correctly and that the updated software operates as intended. This may involve checking for any installation errors, verifying the integrity of the installed software, and monitoring the device's performance post-update. By doing so, they

will ensure that updates do not disrupt device functionality or compromise user safety.

Currently, these Monitoring Agents are implemented using Influx Telegraf[6] and are installed as Docker containers. Influx Telegraf is a highly versatile data collection agent that can gather a wide range of metrics and events from the host devices. Running these agents as Docker containers provides a consistent and isolated environment, simplifying deployment and management across diverse hardware and software configurations. Looking ahead, the goal is to develop more lightweight Monitoring Agents that can perform the same functions with reduced resource consumption.

## 9.5 Summary

The proposed approach leverages Digital Twins to enhance the SSU lifecycle, offering a robust, scalable framework for managing software updates across large fleets of IoT devices. Digital Twins provide a real-time, up-to-date view of each device's status, acting as a central integration point for various tools and processes involved in secure updates. This approach ensures timeliness by using real-time contextual information to deliver updates with minimal disruption and downtime. The use of the Desired-Reported Property pattern guarantees that updates are installed even if devices are temporarily offline, as they are applied once the devices come back online.

The system also offers granular control over the update process, allowing for targeted updates based on the cyber-physical-social context of each device. This ensures updates are distributed to specific devices or groups as needed, offering flexibility and precision. Furthermore, the approach supports scalability, enabling the management of updates across fleets of hundreds or thousands of devices efficiently. The integration of a sandbox environment allows for updates to be tested in a virtual space before deployment, reducing risks.

In addition to facilitating updates, the Digital Twin Platform ensures state persistence, allowing devices to be restored to their desired state following updates, recoveries, or rollbacks. However, it is important to note that this approach is not a complete solution but rather an integration point for other existing tools and methodologies. While it enhances the SSU lifecycle, it can be further extended and complemented by additional tools to provide a more comprehensive solution to

6. https://www.influxdata.com/time-series-platform/telegraf/.

SSUs. This flexibility makes it an adaptable and valuable approach in the dynamic landscape of secure IoT device management.

## References

[1] D. Liberty, W. Ibrahim, M. Streets, B. J. L. Jaeger, M. Wiseman, D. Krahn, G. Proudler, S. Hanna, M. Areno, S. Lee, I. McDonald, D. Challener, A. Seema, P. England, R. Spiger, D. Wilkins, J. Quévremont, Q. Thareau and R. Pal, "TCG Guidance for Secure Update of Software and Firmware on Embedded Systems," 2020.

[2] R. Dautov and H. Song, "Context-Aware Digital Twins to Support Software Management at the Edge," in *International Conference on Research Challenges in Information Science*, 2023.

[3] R. Dautov, H. Song and N. Ferry, "A light-weight approach to software assignment at the edge," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020.

[4] N. Ferry, H. Song, R. Dautov, P. H. Nguyen and F. Chauvel, "Model-based continuous deployment of SIS," in *DevOps for Trustworthy Smart IoT Systems*, N. Ferry, H. Song, A. Metzger og E. Rios, Red., Now Publishers Inc., 2021, p. 59–93.

[5] H. Song, R. Dautov, N. Ferry, A. Solberg and F. Fleurey, "Model-based fleet deployment of edge computing applications," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020.

[6] R. Dautov, H. Song and N. Ferry, "Towards a sustainable IoT with last-mile software deployment," in *2021 IEEE Symposium on Computers and Communications (ISCC)*, 2021.

[7] R. Dautov, S. Distefano and R. Buyya, "Hierarchical data fusion for smart healthcare," *Journal of Big Data*, vol. 6, p. 1–23, 2019.

[8] R. Dautov and H. Song, "Towards agile management of containerised software at the edge," in *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, 2020.

[9] Ayyoob Hamza, Dinesha Ranathunga, Hassan Habibi Gharakheili, Matthew Roughan and Vijay Sivaraman, "Clear as MUD: Generating, validating and applying IoT behavioral profiles," *Proceedings of the 2018 Workshop on IoT Security and Privacy*, p. 8–14, 2018.

[10] Rafiq Ahmad Khan, Siffat Ullah Khan, Habib Ullah Khan and Muhammad Ilyas, "Systematic literature review on security risks and its practices in secure software development," *IEEE Access*, 10, p. 5456–5481, 2022.

[11] Lynn Futcher and Rossouw Von Solms, "Guidelines for secure software development," *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, p. 56–65, 2008.

[12] Mehdi Kherbache, Moufida Maimour and Eric Rondeau, "Digital twin network for the IIoT using eclipse ditto and hono," *IFAC-PapersOnLine*, 55, 8, p. 37–42, 2022.