An Internet of Things Software and Firmware Update Architecture Based on the SUIT specification (WIP)

SIMON CARLSON

Master in Information Technology
Date: May 6, 2019
Supervisor at KTH: Farhad Abtahi
Supervisor at RISE: Shahid Raza
Examiner: Elena Dubrova
Swedish title: En Mjukvaru- och Firmwareuppdateringsarkitektur för Internet of Things Baserad på SUIT-specifikationen (WIP)
School of Computer Science and Communication

Abstract

E

Keywords: IoT, industrial IoT, security, Contiki, embedded systems, software updates

Sammanfattning

Svensk sammanfattning här.

Nyckelord: IoT, industriell IoT, säkerhet, Contiki, inbyggda system, mjukvaruuppdateringar

Acknowledgements

E

Contents

List of Tables

List of Figures

Listings

Acronyms

ACE Authentication and Authorization for Constrained Environments

CA Certificate Authority

CBOR Concise Binary Object Representation

CoAP Constrained Application Protocol

COSE CBOR Object Signing and Encryption

DTLS Datagram Transport Layer Security

EST Enrollment over Secure Transport

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

OSCORE Object Security for Constrained RESTful Environments

PKI Public Key Infrastructure

SUIT Software Updates for Internet of Things

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

URI Uniform Resource Identifier

Chapter 1

Introduction

Internet of Things (IoT) is the notion of connecting physical objects to the world-spanning Internet in order to facilitate services and communication both machine-to-human and machine-to-machine. By having everything connected from everyday appliances to vehicles to critical parts of infrastructure, computing will be ubiquitous and the Internet of Things realized. The benefits from the IoT can range from quality of life services, such as controlling lights and thermostats from afar, to data gathering through wireless sensor networks, to enabling life critical operations such as monitoring a pacemaker or traffic system. IoT is a broad definition and fits many different devices in many different environments, what they all have in common is that they are physical and connected devices.

The Internet of Things is expected to grow massively in the following years as devices get cheaper and more capable. Ericsson expects the amount of IoT connections to reach 22.3 billion devices in 2024, of which the majority is short-range IoT devices [1]. However, cellular IoT connections are expected to have the highest compound annual growth rate as 4G and new 5G networks enable even more use cases for IoT devices.

In recent years the general public has become increasingly aware of digital attacks and intrusions affecting their day to day lives. In 2016 the DNS provider Dyn was attacked by a botnet consisting of heterogeneous IoT devices infected by the Mirai malware. Devices such as printers and baby monitors were leveraged to launch an attack on Dyn's services which affected sites like Airbnb, Amazon, and CNN [2]. Cardiac devices implanted in patients were also discovered to be

unsafe, with hackers being able to deplete the batteries of pacemakers prematurely [3]. These cases and many others make it clear that security in IoT is a big deal, and as IoT is expected to grow rapidly insecure devices can cause even more problems in the future.

In addition to malware there are many other security concerns in IoT networks. These issues include authentication, access control, multi-protocol networking, and updates [4]–[6]. Many security issues are results of the nature of IoT devices, for instance not being able to implement complex security mechanisms on devices due to software or hardware constraints. This means security solutions for IoT must be lightweight and designed specifically with IoT in mind and not traditional networks.

Security in IoT is also closely related to its business potential. According to Bain & Company, the largest barrier for Internet of Things adaptation is security concerns, and customers would buy an average of 70% more IoT devices if they were secured [7]. Despite security being lacking today in IoT, the field is expected to grow rapidly and an increase in unsecured devices could spell disaster. Securing IoT equipment such as printers, baby monitors, and pacemakers is imperative to prevent future attacks. But what about the devices currently employed without security, or devices in which security vulnerabilities are discovered after deployment? They need to be updated and patched in order to fix these vulnerabilities, but sending a technician to each and every of the predicted 22.3 billion devices is not feasible. They need secure and remote updates which is a non-trivial task.

1.1 Problem Statement

There is a need for secure software and firmware updates for IoT devices as vulnerabilities must be patched. For many IoT devices this mean patches must be applied over long distances and possibly unreliable communication channels as sending a technician to each and every device is unfeasible. There are non-open, proprietary solutions developed for specific devices but no open and interoperable standard. The Internet Engineering Task Force (IETF) Software Updates for Internet of Things (SUIT) working group aims to define an architecture for such a mechanism without defining new transport, discovery, or security mechanisms [8]. By expanding upon the work of the SUIT

group, a standardized update mechanism suitable for battery powered, constrained, and remote IoT devices can be developed.

1.1.1 Problem

The thesis aims to investigate the problem "How can the SUIT standard be applied to develop a technology agnostic and interoperable update architecture for heterogeneous networks of Internet of Things devices?". The thesis project will examine the architecture and information model proposed by SUIT in order to create and evaluate an updating mechanism for battery powered, constrained, and remote IoT devices with a life span exceeding five years.

1.1.2 Purpose

The purpose of the thesis project is to provide an open and interoperable update mechanism that complies with the standards suggested by the IETF SUIT working group. This will aid other projects trying to secure their IoT devices following accepted standards.

1.1.3 Goal

The goals of the thesis are to study current solutions and technologies and propose a technology agnostic update architecture based on the SUIT architecture, as well as proposing lightweight end-to-end protocols that instantiates the architecture, allowing updates of IoT devices. The degree project shall deliver specifications and prototypes of the protocols in an IoT testbed, as well as a thesis report.

1.2 Methodologies

The thesis will follow a mix of qualitative and quantitative methodologies. The SUIT group defines some goals or constraints a suitable updating mechanism should follow, but as their proposed architecture is agnostic of any particular technology these goals cannot be easily quantified. It is better to regard them as qualitative properties the mechanism should have. In addition there are some relevant measurements, such as reliability of the communication and memory and power requirements of the updates, that can be used in an evaluation.

The quantitative part of the evaluation will follow an objective, experimental approach, while the qualitative part will follow a interpretative approach.

1.3 Risks, Ethics, and Sustainability

As IoT devices become more commonplace their security becomes more important. Incorrect or faulty firmware or software can lead to incorrect sensor readings, a common application of IoT devices, which in turn can lead to incorrect conclusions. This could have a large effect on businesses such as agriculture and healthcare. Insecure communication channels can expose confidential or personal data, something governments and international unions are taking more seriously.

As these devices might be connected on networks with other computers such as laptops, a compromised IoT device can cause an attack to proliferate throughout an entire network. This can affect other IoT devices as well as traditional computers, putting every connected device at risk. Furthermore hacked devices can leak data to attackers and cause service disruptions. All of these risks and more have to be accounted for as IoT is expected to boom.

By providing updates for devices their life expectancy can increase. As IoT devices are produced in the billions, increasing the lifespan of devices is important from a sustainability perspective. If devices cannot be updated but instead have to be completely replaced often it is a waste of resources. Updating devices is a crucial part in achieving efficiency in how IoT devices are used.

1.4 Scope

The thesis will primarily focus on the use case of applying an entire update to one single device consisting of only one microcontroller. The use case of applying differential updates is considered important and the architecture will enable these kinds of updates, but the prototype will only support it with respect to time.

1.5 Related Work

SWUpdate is an update agent for embedded Linux systems that supports different update strategies, payload formats, custom handlers, and other features [9]. It is released under the GPL-2.0 license and is suitable for open source solutions. As mentioned however, it is only for more capable devices running Linux systems and is thus not compatible with SUIT.

Mender is another software updater for embedded Linux devices released under the Apache license [10]. Mender includes a client as well as management server and offers a paid, hosted solution in addition to their open source code base. As with SWUpdate, Mender targets more capable devices than SUIT focuses on.

MCUboot is a secure bootloader for 32-bit microcontrollers released under the Apache-2.0 license [11]. It aims to define a common bootloader infrastructure as to enable secure software upgrades and is operating system and hardware independent. It does not however deal with transport of updates at all as it is only a bootloader.

1.6 Outline

Chapter 2 will provide the relevant background about the used protocols and operating system as well as the SUIT specification. Following that, Chapter 3 presents the proposed architecture of the thesis, a life cycle approach for IoT updates, examples of profiles for implementing the architecture, and a description of a prototype implementation. Chapter 4 shows the results from evaluating the proposed architecture in both a qualitative and quantitative manner, and Chapter 5 ends the thesis with a discussion, security considerations, and future work.

Chapter 2

Background

Understanding why IoT updates are needed a solution must be proposed. In order to propose a solution however, the current state of the art must be understood. IoT networks use different protocols compared to traditional computing for reasons such as reliability and performance. This chapter presents the background needed to understand the solutions presented in the thesis.

Section 2.1 introduces the network protocols used and motivates their use over other protocols in an IoT context. The following section, Section 2.2 presents and explains the SUIT architecture and information model and their respective requirements as formulated by the IETF. Finally, Section 2.3 presents the Contiki-NG operating system which the updating mechanism will be developed for as well as the target hardware.

2.1 IoT Network Stack

Network protocols in IoT networks operate under different circumstances compared to traditional computer networks. Whereas traditional networks enjoy high reliability, high throughput, and high computational performance IoT networks are defined by their low power requirements, low reliability, and low computational performance on edge devices. This posts some constraints on the protocols used in IoT as they must properly handle these characteristics.

One of the most widely used network protocol stacks today in traditional networks is based on the Internet Protocol (IP). The stack uses Transmission Control Protocol (TCP) as a transportation protocol,

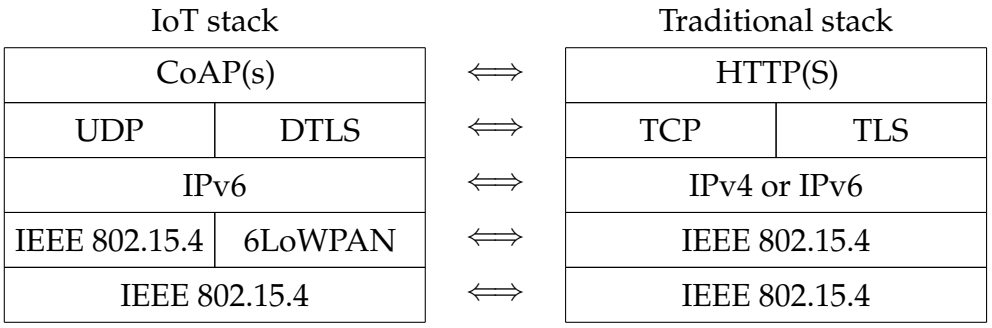


Figure 2.1: Comparison of network stacks between IoT networks and traditional networks.

usually with Transport Layer Security (TLS) for security, and a common application layer protocol is Hypertext Transfer Protocol (HTTP). TCP is however poorly suited for IoT networks as it is a connection based, stateful protocol which tries to ensure the guaranteed delivery of packets in the correct order. There is also advanced congestion control mechanisms in TCP which are hard to apply on unreliable low-bandwidth networks. IoT networks often utilize User Datagram Protocol (UDP) as a transport protocol instead. UDP is also an IP-based protocol but is connectionless and less reliable than TCP, performing on a best-effort level instead. Despite being less reliable, UDP often performs better in IoT contexts.

As TLS is based on the same assumptions as TCP it is unsuitable for UDP networks. UDP networks still need confidentiality and integrity through some means and thus use Datagram Transport Layer Security (DTLS), which is a version of TLS enhanced for use in datagram oriented protocols. HTTP can be used over UDP for the application layer, but as HTTP is encoded in human readable plaintext it is unnecessarily verbose and not optimal for constrained networks. Instead, Constrained Application Protocol (CoAP) is a common protocol for the application layer in IoT networks. Figure 2.1 shows the equivalent protocols for IoT network stacks versus traditional network stacks.

In this chapter, Section 2.1.1 explains UDP and why it is the preferred transport protocol in IoT networks. Section 2.1.2 briefly explains TLS, why it is unsuitable for IoT networks, the differences between TLS and DTLS and why DTLS is used instead. Section 2.1.3 describes the CoAP protocol while Section 2.1.4 briefly introduces OSCORE. Lastly,

sections 2.1.5 and 2.1.6 briefly introduces the EST-coaps protocol and ACE framework, enabling asymmetric cryptography and authorization in an IoT context.

2.1.1 UDP

User Datagram Protocol (UDP) is a stateless and asynchronous transfer protocol for IP [12]. It does not provide any reliability mechanisms but is instead a best-effort protocol. It also does not guarantee delivery of messages. For general purposes in unconstrained environments TCP is usually the favored transport protocol as it is robust and reliable, but in environments where resources are scarce and networks unreliable, a stateful protocol like TCP could face issues. Since TCP wants to ensure packet delivery, it will retransmit packages generating a lot of traffic and processing required for a receiver. Also, if the connection is too unstable TCP will not work at all since it can no longer guarantee the packets arrival. The best-effort approach of UDP is favorable in these situations, in addition to UDP being a lightweight protocol requiring a smaller memory footprint to implement.

2.1.2 DTLS

Datagram Transport Layer Security (DTLS) is a protocol which adds confidentiality and integrity to datagram protocols like UDP [13]. The protocol is designed to prevent eavesdropping, tampering, or message forgery. DTLS is based on TLS, a similar protocol for stateful transport protocols such as TCP, which would not work well on unreliable networks as previously discussed. The main issues with using TLS over unreliable networks is that TLS decryption is dependant on previous packets, meaning the decryption of a packet would fail if the previous packet was not received. In addition, the TLS handshake procedure assumes all handshake messages are delivered reliably which is rarely the case in IoT networks using UDP.

DTLS solves this by banning stream ciphers, effectively making decryption an independent operation between packets, as well as adding explicit sequence numbers. Furthermore, DTLS supports packet retransmission, reordering, as well as fragmenting DTLS handshake messages into several DTLS records. These mechanisms makes the handshake process feasible over unreliable networks.

By splitting messages into different DTLS records, fragmentation at the IP level can be avoided since a DTLS record is guaranteed to fit an IP datagram. IP fragmentation is problematic in low-performing networks since if a single fragment of an IP packet is dropped all fragments of that packet must be retransmitted, and thus fragmenting at the IP level should be avoided. Since DTLS is designed to correctly handle reordering and retransmission in lossy networks, splitting messages into several DTLS records is no problem, and if one record is lost only that record needs to be retransmitted in a single IP packet.

In order to communicate via TLS and DTLS, a handshake has to be carried out. The handshake establishes parameters such as protocol version, cryptographic algorithms, and shared secrets. The TLS handshake involves hello messages for establishing algorithms, exchanging random values, and checking for earlier sessions. Then cryptographic parameters are shared in order to agree on a shared premaster secret. The parties authenticate each other via public key encryption, generate a shared master secret based on the premaster secret, and finally verifies that their peer has the correct security parameters.

The DTLS handshake adds to this a stateless cookie exchange to complicate DoS attacks, some modifications to the handshake header to make communication over UDP possible, and retransmission timers since the communication is unreliable. Otherwise the DTLS handshake is as the TLS handshake.

2.1.3 CoAP

Constrained Application Protocol (CoAP) is an application layer protocol designed to be used by constrained devices over networks with low throughput and possibly high unreliability for machine-to-machine communication [14]. While designed for constrained networks, a design feature of CoAP is how it is easily interfaced with HTTP so that communication over traditional networks can be proxied. CoAP uses a request/response model similar to HTTP with method codes and request methods that are easily mapped to those of HTTP. Furthermore CoAP is a RESTful protocol utilizing concepts such as endpoints, resources, and Uniform Resource Identifier (URI). These features makes it easier to design IoT applications that seamlessly interact with traditional web services. Additionally CoAP offers features such as multicast support, asynchronous messages, low header overhead, and UDP and

DTLS bindings which are all suitable for constrained environments.

As CoAP is usually implemented on top of UDP, communication is stateless and asynchronous. For this reason CoAP defines four message types: Confirmable, Non-confirmable, Acknowledgment, and Reset. Confirmable messages must be answered with a corresponding Acknowledgment, this provides one form of reliability over an otherwise unreliable channel. Non-confirmable messages do not require an Acknowledgment and thus act asynchronously. Reset messages are used when a recipient is unable to process a Non-confirmable message.

Since CoAP is based on unreliable means of transport, there are some lightweight reliability and congestion control mechanisms in CoAP. Message IDs allows for detection of duplicate messages and tokens allow asynchronous requests and responses be paired correctly. There is also a retransmission mechanism with an exponential back-off timer for Confirmable messages so that lost Acknowledgments does not cause a flood of retransmissions. Additionally, CoAP features piggybacked responses, meaning a response can be sent in the Acknowledgment of a Confirmable or Non-Confirmable request if the response fits and is available right away. This also reduces the amount of messages sent by the protocol.

The length of the payload is dependant on the carrying protocol and is calculated depending on the size of the CoAP header, token, and options as well as maximum DTLS record size. Section 4.6 of the CoAP standard says "If the Path MTU [Maximum Transmission Unit] is not known for a destination, an IP MTU of 1280 bytes SHOULD be assumed; if nothing is known about the size of the headers, good upper bounds are 1152 bytes for the message size and 1024 bytes for the payload size" [14].

Since firmware images can be relatively large their size needs to be handled during transportation, which can be done via block-wise transfers [15]. A Block option allows stateless transfer of a large file separated in different blocks. Each block can be individually retransmitted and by using monotonically increasing block numbers, the blocks can be reassembled. The size of blocks can also be negotiated between server and client meaning they can always find a suitable block size, making the mechanism quite flexible. Another option of interest is the observe option, which allows a client to be notified by the server when a particular resource changes. This option can be used in a pull or hybrid update architecture, meaning the device does not have to

continuously poll the server for a new update.

2.1.4 OSCORE

Object Security for Constrained RESTful Environments (OSCORE) is a way of protecting CoAP messages at the application level [16]. When using CoAP secured by DTLS, security is terminated at intermediate proxies meaning data can be eavesdropped or modified. OSCORE avoids this by protecting sensitive parts of CoAP messages, but does not fully protect the entire CoAP message. The mapping capabilities between CoAP and HTTP are not affected by the use of OSCORE and OSCORE protected messages can be translated to use different transport protocols as well. OSCORE is therefore suited to use within constrained networks but also for interfacing with traditional networks while retaining end-to-end security. The OSCORE standard is a work in progress.

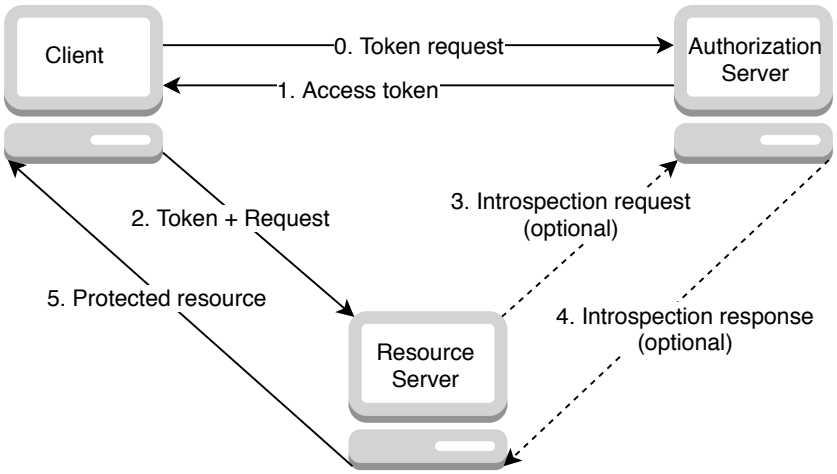
2.1.5 EST-coaps

EST-coaps is a protocol being standardized for certificate enrollment in constrained environments using CoAP. [17]. By allowing IoT devices to enroll for certificates, asymmetric encryption can be used even in a constrained environment. EST-coaps is heavily based on Enrollment over Secure Transport (EST) which was developed for traditional, less constrained networks and is thus incompatible with the SUIT standard, which is specified to work on very small devices [18]. EST-coaps retains much of the functionality and structure of EST but modifies it slightly to work over CoAP, DTLS, and UDP instead of HTTP, TLS, and TCP, for instance by making use of CoAP's block requests and responses to remedy the relatively large sizes of certificates. A corresponding profile for OSCORE is in development [19].

2.1.6 ACE

Authentication and Authorization for Constrained Environments (ACE) is a framework being standardized for authorization and authentication for IoT contexts, based on the OAuth 2.0 framework [20]. ACE allows for clients in a network to access protected resources through authorization tokens.

Figure 2.2: The protocol flow of ACE. Adapted from [20].



A client can request an authorization token from an authorization server and then make a request with the token to a resource server. If the token is valid and the resource requested matches the level of authority associated with the token, access to the resource is granted. The resource server can perform an introspection request to the authorization server if it needs extra information to verify the token. The exchange is depicted in Figure 2.2.

Authorization is important and distinct from identification, which can be achieve through for instance a PKI. As the architecture proposed in this thesis aims to be standardized it should prepare for different parties assuming different roles in the context of updating devices. Using authorization tokens is a scalable and configurable way of achieving that.

2.2 SUIT

The IETF Software Updates for Internet of Things (SUIT) working group aims to define a firmware update solution for IoT devices that is interoperable and non-proprietary [8]. The working group does not however try to define new transport, discovery, or security mechanisms making their proposal agnostic of any particular technology. SUIT aims to define both a mechanism for transporting firmware images as well as meta-data needed to securely update an IoT system.

Section 2.2.1 presents the SUIT architecture and Section 2.2.2 presents

the SUIT information model.

2.2.1 Architecture

There is an Internet-Draft by the SUIT group focusing on the architecture of an IoT update mechanism [21]. Internet-Drafts are works in progress and may be altered or obsoleted at any time, the thesis references the architecture draft issued January 16 2019. This draft describes the goals and requirements of such an architecture, although makes no mention of any particular technology. The overarching goals of the update process is to thwart any attempts to flash unauthorized, possibly malicious firmware images as well as protecting the firmware image's confidentiality and integrity. These goals reduces the possibility of an attacker either getting control over a device or reverse engineering a malicious but valid firmware image as an attempt to mount an attack.

In order to accept an image and update itself, a device must make several decisions about the validity and suitability of the image. The information needed comes in form of a manifest. The next section will describe the requirements posted upon this manifest in more detail. The manifest helps the device make important decisions such as if it trusts the author of the new image, if the image is intact, if the image is applicable, where the image should be stored and so on. This in turns means the device also has to trust the manifest itself, and that both manifest and update image must be distributed in a safe and trusted architecture. The draft [21] presents ten qualitative requirements this architecture should have:

Agnostic to how firmware images are distributed: the mechanism should not assume a particular technology is used to distributed manifests and images, but instead be able to be carried over different mediums. This means decisions about formats and distribution methods must not rely on features of a particular technology.

Friendly to broadcast delivery: the mechanism should be broadcast friendly, meaning the mechanism can not be reliant on security on the transport layer or below. Also, devices receiving broadcast updates not meant for them should not incorrectly apply the update.

Use state-of-the-art security mechanisms: the SUIT standard assumes

a Public Key Infrastructure (PKI) is in place. The PKI will allow for trusted communication.

Rollback attacks must be prevented: the manifest will contain metadata such as monotonically increasing sequence numbers and best-before timestamps to avoid rollback attacks.

High reliability: the act of upgrading an image should not cause the device to break. This is an implementation requirement.

Operate with a small bootloader: the bootloader should be minimal. This is also an implementation requirement.

Small parser: it must be easy to parse the fields of the update manifest as large parser can get quite complex. Validation of the manifest will happen on the constrained devices which further motivates a small parser and thus less complex manifests.

Minimal impact on existing firmware formats: the update mechanism itself must not make assumptions of the current format of firmware images, but be able to support different types of firmware image formats.

Robust permissions: updates must be authorized before they are applied, and different configurations might have different requirements for authorization. The architecture should enable a flexible and robust permission model.

Operating modes: the draft presents three broad modes of updates: client-initiated updates, server-initiated updates, and hybrid updates, where hybrids are mechanisms that require interaction between the device and firmware provider before updating. The thesis will look into all three of these broad classes. Some classes may be preferred over others based on the technologies chosen in the thesis.

The distribution of manifest and firmware image is also discussed, with a couple of options being possible. The two approaches are shown in Figure 2.3 and Figure 2.4. The first figure shows the manifest and image distributed together to a firmware server. The device then receives the manifest either via pulling or pushing and can subsequently download the image from the same server. Alternatively, as shown

Figure 2.3: Distributing both manifest and image through a firmware server.

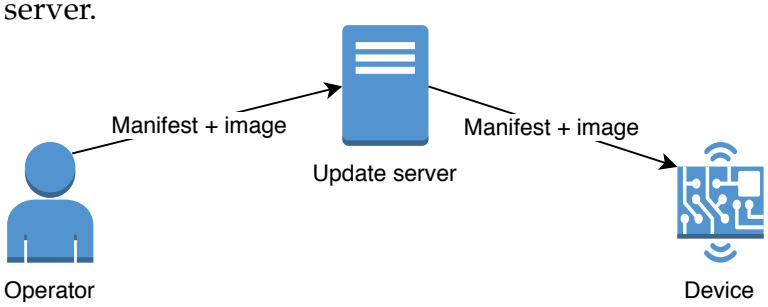
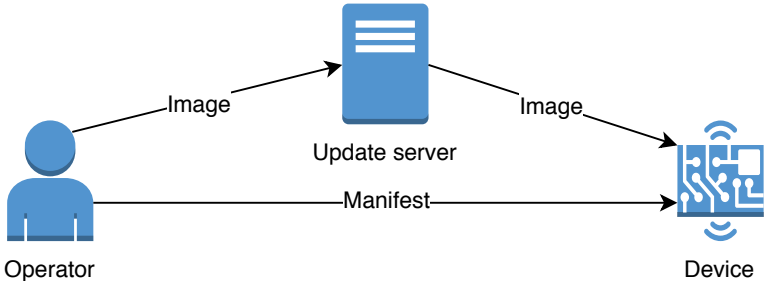


Figure 2.4: Distributing the manifest directly to the device and image through a firmware server.



in the second figure, the manifest itself can be directly sent to the device without a need of a firmware server, while the firmware image is put on the firmware server. After the device has received the lone manifest through some method, the firmware can be downloaded from the firmware server. The SUIT architecture does not enforce a specific method to be used when delivering the manifest and firmware, but states that an update mechanism must support both types.

2.2.2 Information Model

The Internet-Draft for the SUIT information model presents the information needed in the manifest to secure a firmware update mechanism [22]. As mentioned in the previous section, Internet-Drafts are works in progress and the thesis references the information model draft from January 18 2019. A manifest is needed for a device to make a decision about whether or not to update itself, and if the image related to the manifest is valid and its integrity ensured. The draft also presents threats, classifies them according to the STRIDE model, and presents

security requirements that map to the threats [23]. Finally it presents use cases and maps usability requirements to the use cases in order to motivate the presence of each manifest element. Note that the information model does not discuss threats outside of transporting the updates, such as physical attacks.

The proposed mandatory and recommended manifest elements and their brief motivations can be seen in Table 2.1. For the optional elements and more detailed motivations, use cases, and requirements refer to [22].

Table 2.1: The proposed mandatory and recommended manifest elements of the SUIT information model. Adapted from [22].

Manifest Element	Status	Motivation/Notes
Version identifier	Mandatory	Describes the iteration of the manifest format
Monotonic Sequence Number	Mandatory	Prevents rollbacks to older images
Payload Format	Mandatory	Describes the format of the payload
Storage Location	Mandatory	Tells the device which component is being updated, can be used to establish physical location of update
Payload Digest	Mandatory	The digest of the payload to ensure authenticity. Must be possible to specify more than one payload digest.
Size	Mandatory	The size of the payload in bytes
Signature	Mandatory	The manifest is to be wrapped in an authentication container (not a manifest element itself)

Manifest Element	Status	Motivation/Notes
Dependencies	Mandatory	A list of digest/URI pairs linking manifests that are needed to form a complete update
Precursor Image Digest Condition	Mandatory (for differential updates)	If a precursor image is required, this digest condition is needed
Content Key Distribution Method	Mandatory (for encrypted payloads)	Tells how keys for encryption/decryption are distributed
Vendor ID Condition	Recommended	Helps distinguish products from different vendors
Class ID Condition	Recommended	Helps distinguish incompatible devices in a vendors infrastructure

As the SUIT architecture tries to be a standardized solution it must account for different use cases and different combinations of use cases. As a result many of the mandatory and recommended elements are there to enable certain use cases that might not always be relevant. This means certain information must be prepared for in advance even if it is not going to be used in all cases. There is a trade off with flexibility and size, and as the devices of interest are lightweight it is of interest to reduce the size of the manifest as much as possible without limiting use cases. With these two considerations in mind, a manifest for the architecture proposed in this thesis must be designed to facilitate as many different use cases as possible while keeping sizes to a minimum.

To summarize, the SUIT information model proposes to use a signed manifest that is distributed to each device in need of an update through some method. The device then parses the manifest in order to determine if the update is trusted, suitable, and up to date, with many other optional elements such as if special processing steps or new URIs to fetch the images are needed. The model does not make assumptions about technology which is one of the reasons there are optional elements, not all of them are applicable to all solutions. Nevertheless, the architecture and information model together provides a solid base on

which to design a secure update mechanism for IoT.

2.3 Contiki-NG

Contiki-NG is an open-source operating system for resource constrained IoT devices based on the Contiki operating system [24], [25]. Contiki-NG features an IPv6 network stack designed for unreliable, low-power IoT networks. There are many protocols implemented in the stack, this thesis will look at UDP and CoAP secured by DTLS. Beneath IPv6 Contiki-NG supports IEEE 802.15.4 with Time Slotted Channel Hopping [26].

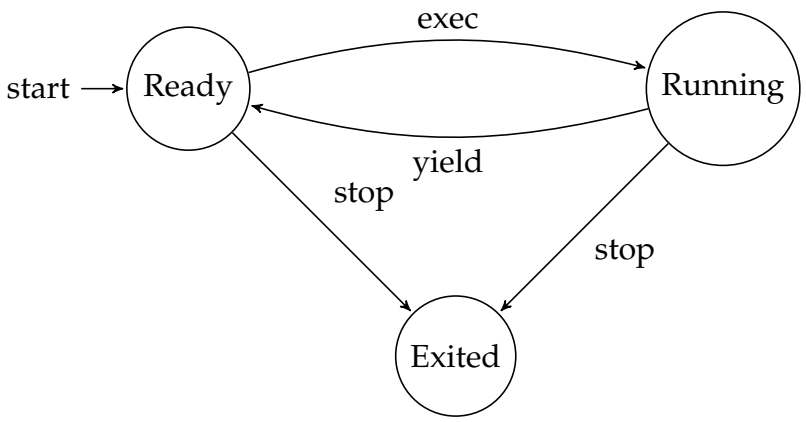
The CoAP implementation in Contiki-NG is based on Erbium by Mattias Kovatsch and supports both unsecured (CoAP) and secured (CoAPs) communication [27]. CoAPs uses a DTLS implementation called TinyDTLS which handles encryption and decryption of messages [28].

Contiki-NG has a process abstraction which is built on lightweight protothreads [29]. Protothreads can be seen as a combination of threads and event-driven programming. Threads provide a way of running sequential programs concurrently, enabling the use of flow-control mechanisms. This however requires a thread-local stack which is too resource heavy on small IoT devices. Event-driven programming on the other hand does not require a thread-local stack but programs are limited to computation as callbacks on event triggers, meaning sequential programs are more difficult to program. Protothreads combine these paradigms in a lightweight way, keeping the yield semantics of threads and stacklessness of event-driven programming. All protothreads in a system are run on the same stack, meaning each protothread has a very low memory overhead, and by providing a conditional blocking wait statement protothreads can execute cooperatively.

Contiki-NGs execution model is event based, meaning processes often yield execution until they are informed a certain event has taken place, upon which they can act. Figure 2.5 shows the states of the processes and their transitions. User-space processes are run in a cooperative manner while kernel-space processes can preempt user-space processes. Examples of events are timers expiring, a process being polled, or a network packet arriving.

Contiki-NG provides two memory allocators in addition to using

Figure 2.5: The state machine of Contiki threads. Adapted from [30].



static memory. They are called `MEMB` and `HeapMem` and are semi-dynamic and dynamic, respectively. `MEMB` is a semi-dynamic allcator that allocates pools of static memory as arrays of constant sized objects. After a memory pool has been declared, it is initialized after which objects can be allocated memory from the pool. All objects allocated through the same pool have the same size. `HeapMem` solves the issue of dynamically allocating objects of varying sizes during runtime in Contiki-NG. It can be used on a variety of hardware platforms, something a standard C `malloc` implementation could struggle with. With `HeapMem` memory can be reallocated and deallocated as if using a normal `malloc`.

2.4 Summary

This chapter has introduced and motivated the use of network protocols common in IoT networks. The SUIT standard for providing updates, which is agnostic of any specific technology, was also introduced. This standard can be expanded and profiled in order to propose a possible solution. The thesis will develop a prototype of such a solution using the Contiki-NG operating system. The next chapter will expand upon the works of SUIT discussing topics such as roles, authorization, and communication of updates, apply a life cycle view to IoT updates, and define profiles for implementing an update architecture.

Chapter 3

Method

3.1 Proposed Architecture for Secure Internet of Things Software Updates

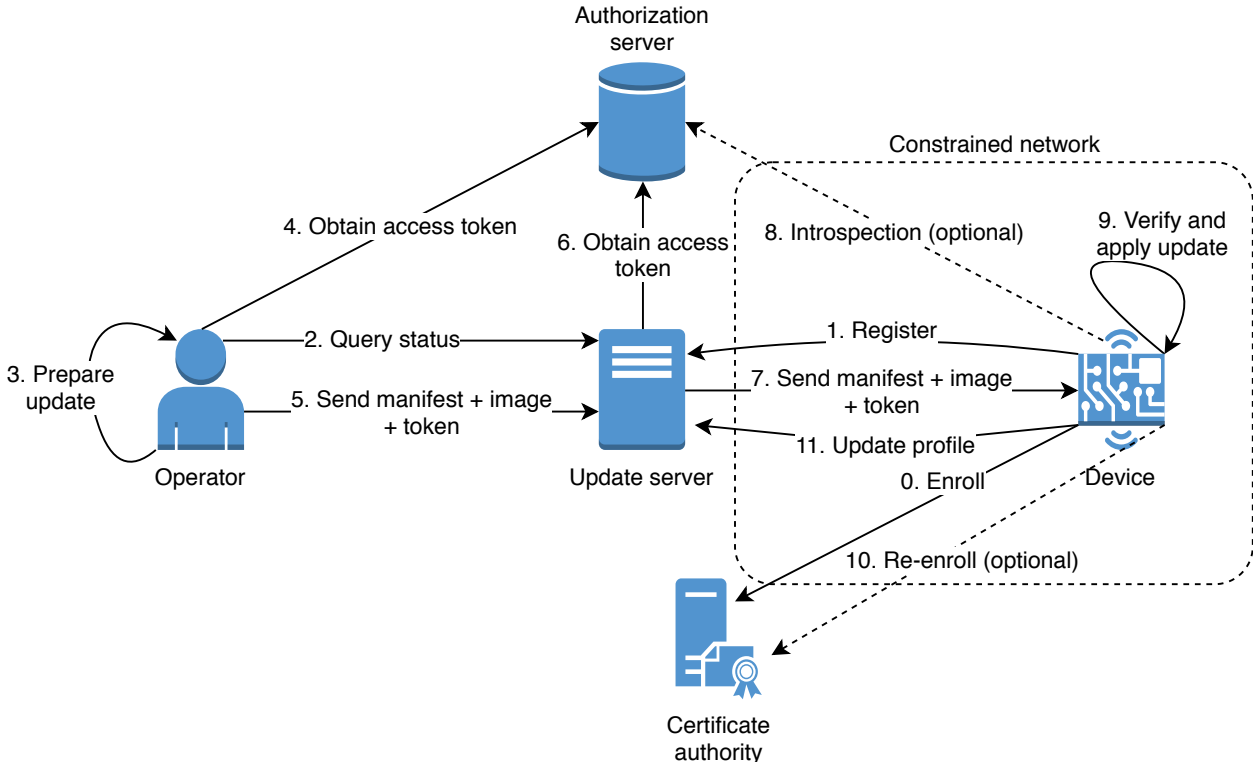
The previous chapter provided the necessary background information to understand the architecture proposed by the thesis in this section. The SUIT architecture standard identifies requirements an update architecture should have and provides minimal examples. From this standard an architecture, still abstract and technology agnostic, can be designed, providing information on topics such as identification, authorization, and device life cycles.

The proposed architecture is technology agnostic and adds on top of SUIT certificates and tokens for identifying and authorizing entities and defines what information is needed for a device to take part of an update procedure. Five key areas that the architecture must define have been identified:

- Roles of devices, servers, and operators
- Key management of IoT update procedure
- Device profiles and communication
- Update authorization
- Update handling for local upgrades

By putting the five key areas together, Figure 3.1 shows an example update flow in the architecture. Note that where the SUIT standard

Figure 3.1: Example workflow of an update procedure.



would differentiate between the author of an update and an operators, this architecture will for simplicity regard them as the same entity, the operator. For the same reason, the architecture will not differentiate between device and network operators. An operator is an entity that tracks device statuses and prepares, signs, and sends updates to devices. Figure 3.1 gives a brief introduction to the concepts needed to understand the architecture, these concepts will be explained in more detail throughout sections 3.1.1-3.1.5. In this particular example the update procedure is initiated by the operator by querying the update server for device status and then pushing an update. The manifest and image are distributed together. The steps can be briefly explained as:

0. The **device** enrolls at the **CA** and receives a **certificate**.

1. The device registers at the **update server** which creates a **profile** for the device.

2. The **operator** queries the update server for device status in order to prepare an update.

3. The operator prepares a **manifest** and **image** and signs them.
4. The operator requests an **authorization token** from the **authorization server** in order to gain access to applying the update.
5. The signed manifest and image are sent to the update server with the authorization token.
6. The update server requests an authorization token from the authorization server in order to gain access to applying the update.
7. The signed manifest and image are sent with the update server's authorization token to the device.
8. The device requests introspection data from the authorization server to verify the authorization token. This step is optional.
9. The device decrypts and verifies the manifest and image and applies the update.
10. The device's certificate broke as part of the update and it uses a new pre-shared key included in the updated image to re-enroll. This step is optional.
11. The device updates its profile at the update server by re-registering.

Section 3.1.1 defines what a device, update server, and operator means in the context of the update architecture. Section 3.1.2 discusses what is needed for devices to enroll in a PKI and how keys are handled during updates. Section 3.1.3 describes how devices, update servers, and operators can communicate and shows an example workflow of communicating an update. Section 3.1.4 describes the purpose of issuing authorization tokens, how devices can use them, and who is to be authorized. Section 3.1.5 discusses different means of handling the payload when applying an update. Section 3.1.6 introduces the manifest format proposed by the architecture. Finally Section 3.1.7 shows the architecture applied to different use cases.

3.1.1 Roles of Devices, Update Servers, and Operators

This section explains the notion of devices, update servers, and operators in the architecture, their responsibilities, and their intended

functionality.

What Is a Device and What Do They Do?

Devices are constrained, low-power IoT appliances connected to a constrained network. They are running the applications of the network and performs simple tasks such as measurement using sensors. The devices communicate wirelessly and must be secured from attackers while being able to be updated. They communicate with each other, update servers, and operators.

Devices have to trust update servers and operators wanting to send them updates. In order to do so, devices need predetermined whitelists of update servers and operators. The identity of an update server or operator can be checked using certificates and by then matching against the whitelists, communication can be allowed or prohibited. Update servers and operators that are whitelisted still need to be authorized to perform an update, this is discussed further in Section 3.1.4.

Communication with update servers and operators requires the update servers and operators know how to reach the devices if they want to initiate contact. Devices must thus register profiles at the update servers, and must be enrolled in order to be trusted. In order to register a profile at an update server, the update server must offer an API that allows a device to POST its information (vendor and class ID as well as version) to the update server. How communication is handled and a further explanation of profiles is in Section 3.1.3.

To properly handle updates a device needs to implement a local update handler. The update handler is responsible for decrypting manifests and images, parsing and verifying the manifest, and preparing the image for boot. How and where the device stores manifest and image is discussed in further detail in Section 3.1.5.

What Is an Update Server and What Do They Do?

Update servers are responsible for transporting manifests and images to the devices, acting as image repositories, and keeping track of device profiles. After enrollment devices register at an update server and the update server creates a profile for that device. The profile contains the vendor and class IDs and firmware version of the device. This allows the update server to know through which protocols the device can be contacted and which version it should be updated to. Operators,

discussed in the next section, send signed manifests and images to update servers, and can query them for device status.

Devices must contain a list of update servers and by trusting the certificate authority they can verify update server certificates. An update server is thus any machine that is enrolled, has a valid certificate, and is included in the device's list of update servers. The reasoning behind this definition is that a standard solution for updates should not assume the topology of an IoT network. The update server may be a machine acting as a proxy between a traditional network containing the operator and a constrained network with IoT devices. The update server could be a more capable IoT device located entirely within the constrained network and be contacted through a proxy. The update server could be located entirely within the traditional network and use a proxy to communicate with devices of the constrained network. Section 3.1.7 shows different use cases, highlighting this aspect.

An update server must provide an API for a device to register, this can be done by letting the device POST vendor and class IDs and version to some registration endpoint. Furthermore devices must be able to pull updates from the update server, this also requires some API the device can use for querying. The querying could be a GET request from the device upon which the update server matches the profile of the device with available manifests. The update server should offload the device as much as possible when it comes to picking an update as devices will typically have much tougher energy constraints. In addition, update servers act as repositories for updates and have much more information regarding which update is suitable.

Additionally operators will also be querying the update server for statuses of devices. This functionality can be more fleshed out as it interacts with a human being and can offer filters for retrieving statuses of devices from a certain vendor or class, or devices running a certain firmware version. Allowing the operator to filter devices in the network while querying is important as IoT networks can contain a large amount of different devices, and it is imperative to find the ones in critical need of updates first.

A device can be aware of several update servers, and different devices can be mapped to different update servers. Optionally, different devices can be mapped to different endpoints of the same physical update server. This can make device management easier as certain classes of devices can be handled by certain update servers. By allowing a

device to receive updates from several update servers, the update mechanism architecture also displays a form of robustness. If one update server is for instance located entirely within the constrained network and the connection between that update server and the operator is severed, updates can still be distributed through other update servers. If devices are pulling updates, they can query the update servers in order of their list of update servers. If updates are pushed, devices keep the connection with the update server pushing the update.

Which machines are allowed to act like update servers can be boiled down to a few important points no matter the topology of choice:

- An update server is enrolled and has a valid certificate
- An update server is included in a device's list of update servers
- An update server can request authorization tokens and be authorized to update a device
- An operator can reach the update server and is authorized by the update server to query device status and upload manifests and images

What Is an Operator and What Do They Do?

Operators are people authorized to upload manifests and images to an update server. They can also optionally upload manifests directly to a device depending on the network topology. Operators prepare manifests and images, signs and transports them with an authorization token to an update server, which then forwards them to a device. The signing ensures end-to-end security for images and manifests between operators and devices. Authorization is further discussed in Section 3.1.4.

In addition to a list of update servers, devices also need a list of operators. This is because if an operator wishes to send a manifest directly to a device, the device needs to be aware of the operator and permit traffic from that operator. Just like with mapping devices to update servers, devices can receive manifests from several operators and different devices may interact with different operators.

Mapping devices to different operators creates opportunities to logically divide a network between operators. An example use case is a

constrained network supported by different vendors where the respective vendors should only be able to service their respective devices. It can also create a hierarchy, where certain operators may directly interact with devices but other operators must interact with devices through update servers. Yet again the point is to prepare for as many different scenarios as possible and create a flexible architecture.

Operators need a way of creating manifests in order to update devices. In order to create a manifest they must be aware of which devices are going to be updated and what firmware versions they are using. This information is held by the update server in form of device profiles. An operator can query the update server selecting the devices of interest based on vendor or class ID or firmware version.

As the manifest contains data difficult for humans to craft such as image hash digest and sequence numbers possibly in the form of timestamps, operators should be able to provide information such as manifest version, vendor and class IDs, firmware version, and URL to a program which then generates, encodes, and signs the manifest for them. Such a program can also sign the image and send the manifest and image without further intervention from the operator, this is implementation specific. The point is operators need help from the update server when deciding about which devices to update, and should be given help filling in the details of the manifest to ensure its correctness.

As with update servers, the essence of being an operator can be captured in a few points:

- An operator is enrolled and has a valid certificate
- An operator is included in a device's list of operators
- An operator can request authorization tokens and be authorized to update a device
- An operator is trusted by an update server to query device status and send manifests and images to the update server

3.1.2 Key Management of IoT Update Procedure

The architecture will, in order to align with the goals of SUIT, be based on asymmetric cryptography. The availability of EST-coaps makes this feasible in IoT contexts but other enrollment protocols could also be used. This means a Certificate Authority (CA) is needed to act as a

trusted third party distributing certificates. The certificates are linked to a public key which has a private key partner and are used to verify the correctness of a public key. Certificates are signed by the CA and in order to trust them, device have to trust the CA itself.

In order to enroll, a pre-shared key is proposed. This is the approach used in EST-coaps, but it is not chosen for that reason. If a pre-shared key is not used the CA cannot be sure the device asking to enroll really should be part of the trusted network. If an attacker obtains a valid certificate they could communicate with devices and update servers alike and no one would be able to tell it is a malicious actor. CAs must know they are issuing certificates to the correct devices and pre-shared keys gives devices a means of identifying themselves. Pre-shared keys could be used for encrypting all traffic but as they are less scalable and harder to manage than certificates, they are just used for enrolling.

A device that is enrolling has to trust the CA issuing the certificate. If it cannot do so, how could it know it received a valid certificate? An attacker would love for a device to use the attacker's public key instead, and if an attacker poses as a CA it could issue a certificate with its own public key and then sign it. In order to trust the CA, a device needs to have the certificate of the CA it is enrolling with or some other CA further down the chain of trust. By verifying the signature on the newly enrolled certificate with the CAs own certificate, a device can be certain they are using the correct keys.

There are options concerning how many key pairs a device should have. Device-to-device communication might occur in the constrained network depending on the applications running and it would also require asymmetric cryptography for identification. Devices receiving updates might communicate with update servers or operators outside the constrained network. If the same key pair is used for device-to-device communication and updates and an attacker gets ahold of that key they can use it both inside and outside the constrained network. Using the same key pair for device-to-device communication and updates makes the consequences of losing keys greater.

Some devices might not be able to handle different key pairs and therefore certificates, either as a result of their constrained memory sizes or as a consequence of not having access to abstractions such as a file system. Using the same key pair for all kinds of device communication is acceptable but should be carefully considered. If many key pairs can be used on the same device, an implementer should consider the

granularity of the key pairs' scope; should they be used per device, per service, or per application?

After applying updates, certificates might not be valid anymore. This is implementation dependant and might not always be a problem, but the architecture should be prepared for these situations. In the case an update breaks a certificate, the certificate cannot be used for communication and thus the device cannot use it to re-enroll either. In this case, a new pre-shared key should be part of the update so that the device can enroll as if it was factory new. The process of issuing pre-shared keys might be difficult to automate as the CA must be aware of which keys to accept, but it is needed to ensure the security of future device communication.

3.1.3 Device Profiles and Update Communication

In heterogeneous networks of IoT devices, each device might sport a specific protocol stack. In order to enable different devices to be updated, update servers must be aware of how to reach these devices. This problem introduces the notion of device profiles containing information about device capabilities and software versions.

When devices have enrolled and obtained a valid certificate, they must contact their respective update servers so the update servers can create profiles of the devices. The profiles will tell through which protocols to reach a device and what software versions the device is using. In order to achieve this, devices must first know which update servers to contact. This can be solved through shipping devices with a list of update servers. This list can later on be updated like any other software. Furthermore, the SUIT information model uses vendor and class IDs to verify an update is intended for a specific device by matching the IDs. These IDs can also be sent to an update server to tell it what kind of device is contacting it. The update server can infer a profile based on the IDs it is sent, or simply use the protocols the device chose to contact it. The devices also need to be aware of how to register, for instance by POSTing to a specific update server registration endpoint.

When updating there are possibilities regarding how the updating process is initiated. An operator can query the update server for the status of one or several devices, prepare an update for them, and have the update pushed through the update server. Optionally the operator

could send a manifest to the device explaining when the update is to be applied and put the image on the update server. Later when the device should update, the device pulls the image from the update server, verifies it using the already received manifest, and updates. Both the pull and push approaches assume the devices are already enrolled and registered at the update server.

After updating, the device's capabilities might have changed, for instance by having a new protocol implemented in software. The version of the device will also have changed due to applying an update. After applying an update, a device should notify all its update servers so they can update the device profile (or simply discard the old one and generate a new, as if the device registered for the first time). This ensures the update servers view of the devices is up to date and that communication will always happen through the intended and supported protocols. The functionality of re-registering is the same as when a device is new and thus not costly to implement.

There are alternatives to using device profiles. One alternative is to instead keep a list of known protocols implemented by devices in the network, and when pushing updates to a device trying each protocol in sequence. This has the benefit of not needing to keep and continuously update profiles, but also has some issues. One issue is that you still need to keep some state of the devices on the update server regarding firmware version. If the update server does not know what the status of device versions are, it cannot help a human operator decide about deploying updates.

Another drawback is that devices might implement common protocols but have different preferences. If two devices implement some common protocols but one of them supports hardware operations for encrypting one of the protocols, it will prefer using that protocol with the update server, whereas the other device might not. The update server will however, without information about device preference, try the same sequence of protocols with both devices.

Furthermore, as communication can be unreliable over these networks, the update server cannot know for sure if a device does not respond due to not understanding the protocol and therefore dropping the packets, or if the response just got lost in transmission. It is more robust to keep track of which protocols devices support and conform to the preferences of the constrained devices.

Lastly, instead of using profiles all communication could be initiated

from the device side, meaning updates are only done through a pulling mechanism. This would not enable the use case of pushing updates which could be critical if a vulnerability must be patched right away. Operators must be given the choice to push updates to their devices, and thus update servers must be able to initiate contact with devices.

3.1.4 Update Procedure Authorization

A flexible architecture enables different configurations of update servers, operators, and devices. An operator might be authorized to update all parts of all devices, or be constrained to updating a specific application for a subset of the devices. Operating system vendors might be allowed to push security updates for the operating system but not change the application code. Controlling access rights is a security issue and the architecture must support it.

In the context of the update architecture, the client would be a party needing to authorize themselves, i.e. a device, operator, or update server. The access token would have to be of a lightweight variant since they are to be used on constrained devices, but which variant is implementation specific. Tokens will be sent with signed manifests and images to devices and the resource sought for is the ability to update the device. The device will thus not respond with a particular resource but instead if the token is valid recognize the update as an authorized one and proceed with verifying it.

Devices will only allow communication from trusted sources specified in predetermined whitelists of update servers and operators. This tells devices which sources of communication to allow which in itself may be a form of access control in certain deployments. For simple devices, for instance devices only running a single service using one microcontroller and a single means of storage, any update received might target the entire device. In this case, controlling update authorization using whitelists of update servers and operators might be sufficient as an allowed update can only update the entire system at once with no further granularity. However, for more complex devices or applications this will not suffice. If different applications, microcontrollers, or storage locations can be targeted, an author or update server must use tokens to access those specific resources they are supposed to access.

Frameworks like OAuth 2.0 and ACE use proof-of-possession tokens that are bound to a cryptographic key. When receiving a proof-of-

possession token, a receiver must verify that the key bound to the token is held by the client sending the token. This means that if an operator sends an update with a token to an update server which forwards the update and token to a device, the device must be able to verify with the *operator* that it holds the key bound to the token and not the update server. The device will in essence not be able to verify this as it received the message from the update server, and not the operator.

In order to remedy this, the operator's authorization token will be used to authorize the communication with the update server and then terminate. If the token is valid, the update server will store the update then ask for a new token for itself which is used for transporting the update to the device. The device can then verify the token's key with the update server which should be the holder of that key. What about when operators send manifests directly to devices and when devices pull updates from the update server? There are four scenarios where tokens must be requested from the different parties:

- When an operator sends a signed image (and possibly manifest) to an update server
- When an operator sends a signed manifest to a device
- When an update server sends a signed image (and possibly manifest) to a device
- When a device pulls an update from an update server or wants to register

Enforcing authorization is a security concern and the parties must be able to both request tokens and verify them if necessary in that deployment. Because proof-of-possession tokens bound to cryptographic keys may be used, the tokens should be used for single hops when sending an update. If not, the receiver of such a token will not be able to verify the holder of the key bound to the token. Authorization enables operators, update servers, and devices to send and receive updates and enables use cases such as differential updates. If a device is running several applications and an operator is only supposed to update one of them, the scope of the update's authorization token can be for that application only. Any attempt to update other applications would fail due to authorization errors. Updates that affects the entire code of the device, i.e. writes an entirely new image, also need to be

authorized as they are changing the code of the device, however for certain deployments this can be done using only whitelists.

3.1.5 Update Handling for Local Upgrades

When an update has finally arrived to the device, it must be processed and then installed. The process of decrypting, verifying, and installing the image is heavily implementation specific with most of the details being out of scope for the architecture. However, since devices operate using different hardware and bootloaders they must be given the freedom to update in the way that makes most sense for them and the architecture should support this while requiring the approach is secure.

By including optional fields such as decryption instructions, processing steps, and postconditions in the manifest, update handlers and bootloaders can take care of updates in various ways. This enables different ways of applying updates within the same architecture. What the update handlers and bootloaders of all devices have in common is that they must trust the source of the update, they must be able to decrypt the update, they must be able to verify the validity of the update, and they must be able to do this safely such that an unexpected power cycle does not brick the entire device.

Still being in the realm of very constrained devices, the bootloader should be as simple as possible. This is also a requirement stated by SUIT on the architecture. By decrypting and verifying the image at every boot the boot procedure will be secure but quite complex. Decrypting the image as part of the update mechanism and writing it unencrypted to flash would allow for a simpler bootloader, but is less secure as the device would boot from an unencrypted image.

By storing the manifest containing the image digest, unencrypted images can be verified by comparing their digest to the one in the manifest. Calculating a digest is less demanding than decryption, and a power cycle would just interrupt the digest calculation instead of the decryption of the actual bootable image. In order to make sure the manifest is correct upon boot, the digest of the manifest can be calculated and stored alongside the manifest and image. Optionally the manifest can be encrypted instead as it typically will be much smaller than the image, but again a power cycle during decryption could lead to undefined behavior.

In addition to storing the manifest for verification upon boot, the

manifest also has to be stored in order to prevent rollback attacks. It contains monotonically increasing sequence numbers to ensure devices do not install older, possibly vulnerable images, but in order to compare sequence numbers the most recent manifest has to be kept around. Manifests should be kept for the update handler to compare sequence numbers, and for the bootloader to verify images.

3.1.6 Manifest Format

Introduced in Section 2.2.2, the manifest provides metadata about the update and allows devices to ensure they are the intended recipients of an update and that it can be trusted. Installing unauthorized or incorrect payloads can lead to a compromised or broken system. The manifest doubles down as a security mechanism preventing malicious or mismatched updates while enabling functionality such as running update handlers at specific times or describing a differential update. It is important that devices are able to parse the manifest efficiently as a large parser goes against the goals of SUIT since it can easily become complex and too large to handle for constrained devices. To use a simple parser the format itself must be simple.

SUIT mentions elements of the manifest but does not specify a format nor implementation examples. When considering a format, certain points should be kept in mind:

- The manifest should be as small as possible
- The manifest should be easy to parse
- It should be possible to add new elements to the manifest as this can enable certain use cases for niche deployments

The manifest must be compact and simple while being flexible enough to allow for new additions. This calls for a baseline manifest containing the mission critical information that is required in most any update, while severing non-critical information to be implemented as options. In order to comply with the SUIT standard, elements that SUIT considers mandatory can be placed in the baseline manifest and elements SUIT considers optional placed in the options. A parser implementation will thus always implement the functionality needed to parse the mandatory elements while implementing support for the

optional elements is up to the implementer. One such example is the payload option. The baseline manifest will contain information about the payload such as size and format, but certain deployments might find it useful to embed the image payload itself in the manifest. This could be the case for differential updates of very small images where the update code is small enough to fit. For these deployments, implementers can implement the optional payload option which a client will use instead of issuing another request to the server to fetch the image.

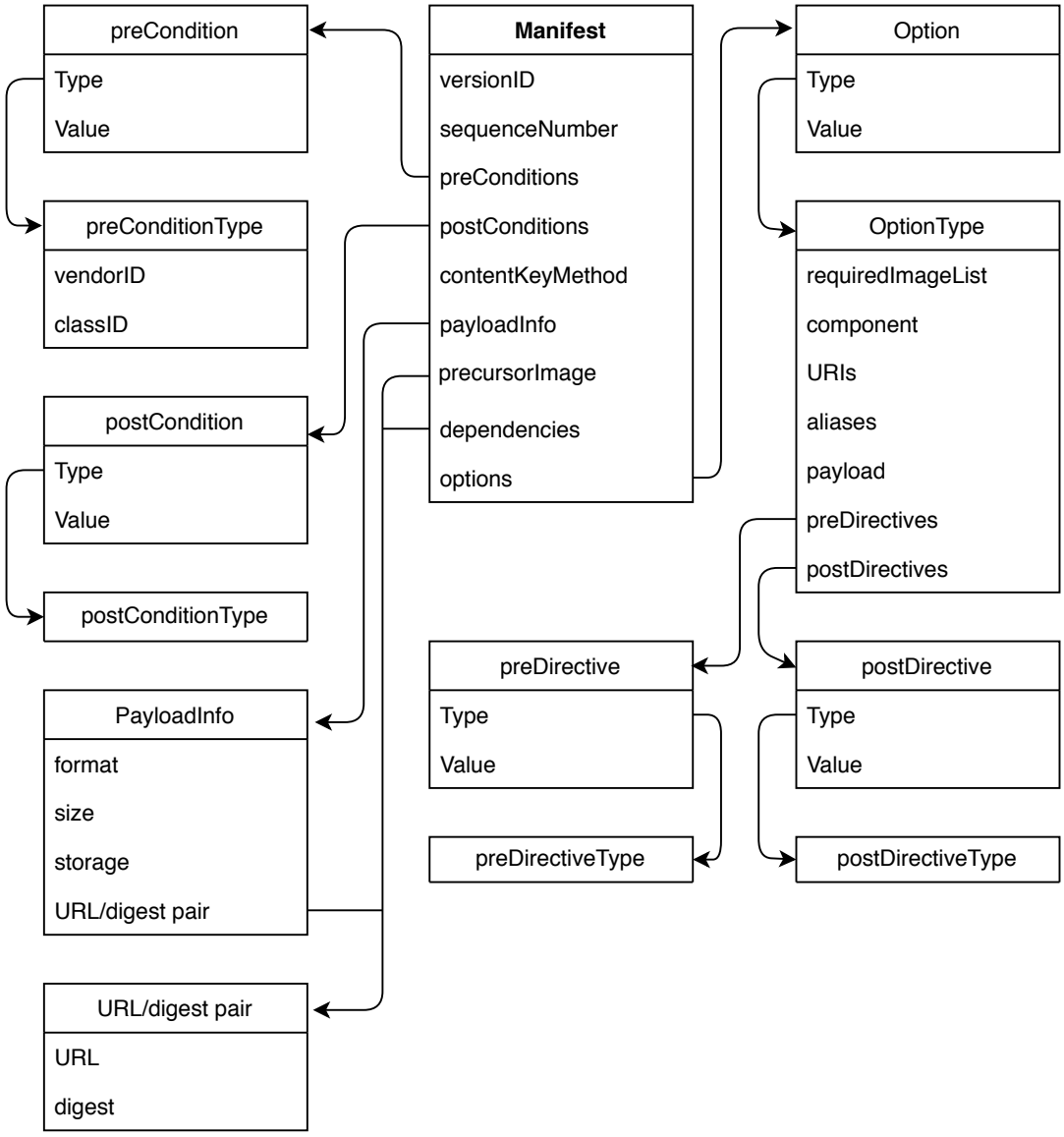
Figure 3.2 shows a schematic view of the manifest format this thesis proposes. The baseline manifest contains version ID, sequence number, precursor images, dependencies, payload information, key method, conditions, and options. The current option types are required image list, component identifier, URIs, aliases, and directives.

Some of these elements are singular elements such as version ID and sequence number, they are simply numbers that are bound to some upper size depending on packet structure. Their sizes are deterministic and they are easy to parse. Other elements, for instance payload information and conditions, have dynamic sizes. This is because the amount of information needed for one update can differ quite drastically from another update. One update without dependencies that is targeted for one specific class of device with no optional elements can be described in a very small manifest. This small manifest would have no precursors, dependencies, or options. The payload information would contain just one URI/digest pair, and there are only two conditions, one vendor ID and one class ID.

A different update that is specifying differential updates for a wide class of devices will need to specify much more information in the manifest. There will be several class IDs as several classes are targeted. The different classes might not be running the same versions of software or firmware and thus several dependencies are listed. Furthermore since devices might not be able to reach the same servers, the manifest contains a list of aliases for the image, causing the manifest to contain many URI/digest pairs. This manifest will be larger than the previous one described and the fields of this manifest will have much different sizes. The dynamic nature of certain elements must be considered when implementing the manifest as the packet structure must be prepared for it.

The condition type, directive type, and option type structures pro-

Figure 3.2: The proposed manifest format.



vides flexibility to implementers. The manifest format as it is specifies the optional elements of SUIT as option types as well as vendor ID and class ID for condition type. It is easy to add new types and thus add new conditions, options, or directives. Conditions are either parameters that must match some property of the device or invariants that must be true before or after the update, while directives provide instructions as how to unpack the update, how to install it etc if so needed. Conditions and directives are split into pre/post variants to avoid erroneously adding for instance vendor ID as a post condition. The structure of pre- and postconditions are the same being separated only for stronger semantics, the same for pre- and postdirectives.

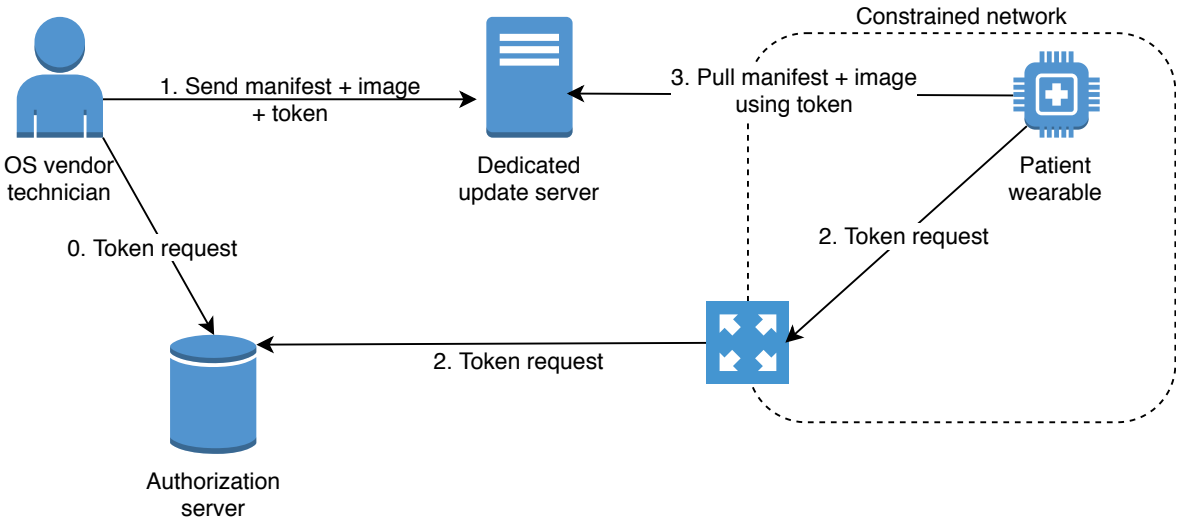
The meaning of options are completely dependant on the specific option in mind, so new functionality can be embedded as an option. If the current manifest format lacks some functionality for a specific deployment, an implementer can add that piece of functionality in the form of an option, updating the server preparing the manifest as well as the parser on the device. Section 3.4.1 discusses the manifest implementation used in the thesis.

3.1.7 Use Cases and Example Topologies

As discussed, operators, update servers, and devices can interact in many ways. The architecture tries to be flexible allowing for different network topologies and configurations. The important parts are that devices can enroll and register, all entities have valid certificates, and that updates can be authorized. The following examples all assume devices are enrolled and registered, and re-enrollment and updating profiles are omitted to make the figures clearer.

Figure 3.3 shows a use case where wearables of patients in an elderly home need their operating system updated. A technician from the operating system vendor assumes the role of an operator. The technician interacts with an authorization update server and a dedicated update server to send the signed manifest and image with an authorization token. The update server accepts the update and stores it for future use. When the device is about to pull the update, it first requests an authorization token. The authorization server does not handle the protocols used inside the constrained network and thus the token request goes through a proxy on the edge of the network. After the device has received the token, it pulls the signed manifest and image using the

Figure 3.3: An operating system vendor technician updates patient wearable in an elderly home.

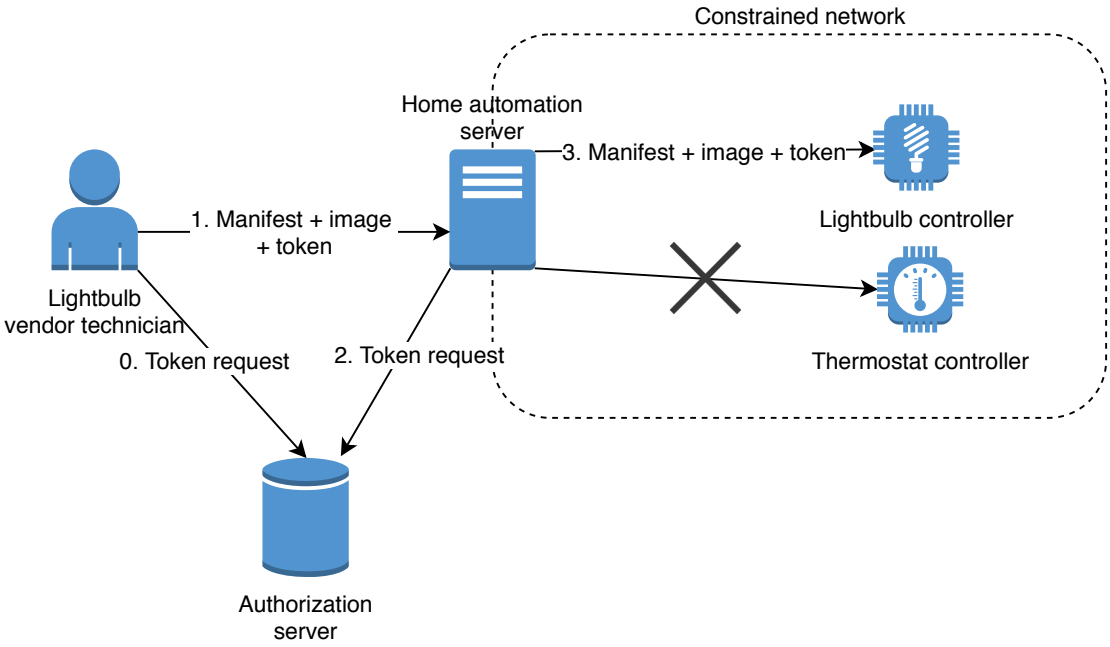


token to authorize the pull.

In Figure 3.4 a technician from a smart light bulb vendor is updating a light bulb controller in a smart home. The technician requests a token and sends it together with the signed manifest and image to a home automation server living on the edge of the constrained network. The server requests a token of its own and pushes the updates to devices in its network. Both a light bulb controller, the intended target, and a thermostat controller receives the update. The thermostat controller is supposed to only accept updates from the thermostat vendor, thus the light bulb vendor technician is unauthorized to perform this update. The thermostat controller rejects the update while the light bulb controller accepts it and updates itself.

The last example shown in Figure 3.5 shows an on-site devops engineer wanting to update a critical temperature sensor in an industry plant. The engineer first requests a token to send the signed manifest directly to the device and a token to send the signed image to the sensor update server. In this case, the update server is a more capable IoT device in the constrained network. It needs to request an authorization token to push the update to the device but is unable to speak the protocols used by the authorization server. The update server instead goes through a proxy to send the token request. With the token, the update can be sent to the device.

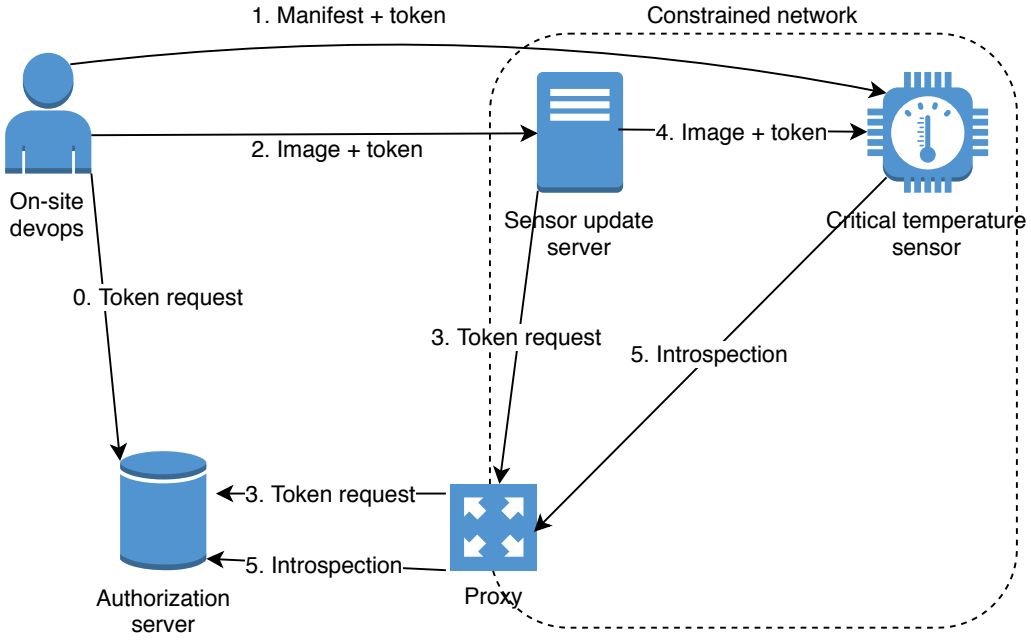
Figure 3.4: A light bulb vendor technician updates a light bulb controller in a smart home, but the thermostat controller rejects it due to authorization issues.



The device needs to authorize the token, but as it is a very constrained device it cannot perform the necessary calculations. The device opts to perform an introspection request through the proxy so the authorization server can help verifying the token. After the device has received an answer, it can proceed applying the update.

What these three examples have in common is that the operator always resides outside the constrained network, devices always reside within the constrained network, and that update servers transport updates to devices alongside authorization tokens. Since operators reside in traditional networks using traditional protocols such as HTTP over TCP, a proxy may be necessary to communicate with the often UDP based constrained networks. The update server can act as a proxy, or a dedicated proxy could be used. If the operator has the proper protocols implemented, they can communicate manifests directly to a device.

Figure 3.5: An on-site devops engineer updates a very constrained, critical temperature sensor in an industry plant. The constrained sensor asks for introspection data in order to verify the update server's token.



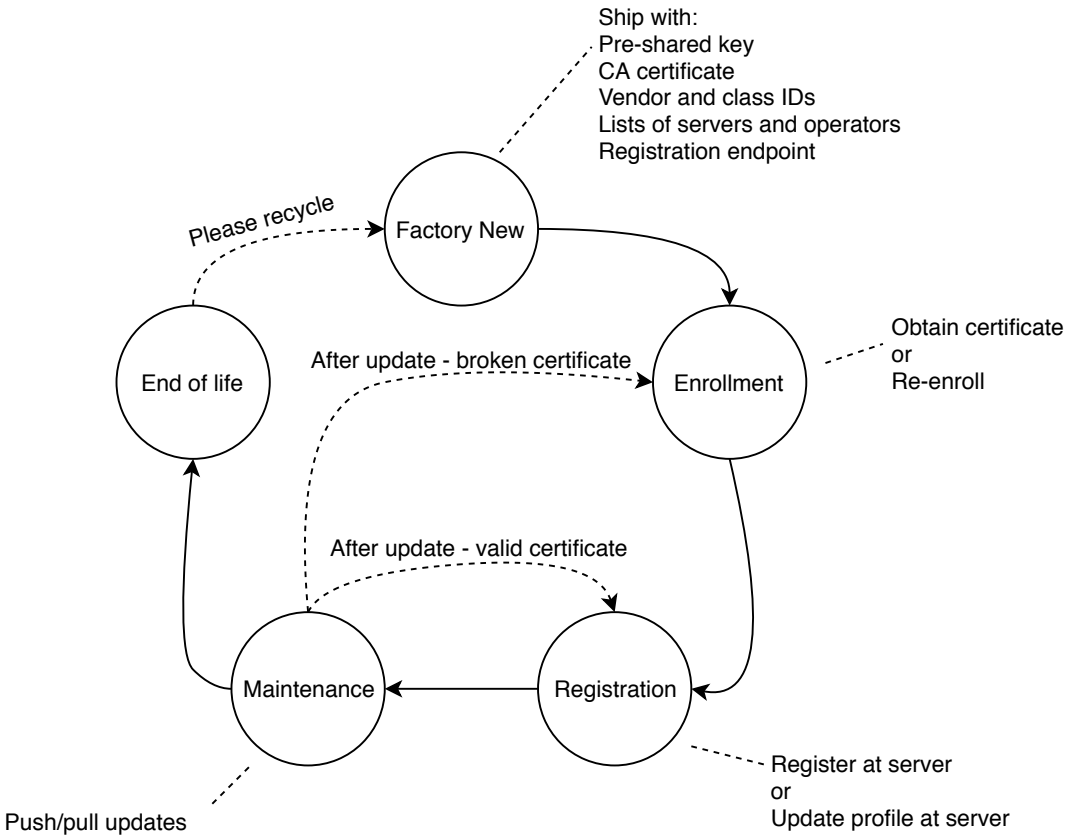
3.1.8 Summary

This section presented the proposed architecture of the thesis. The architecture added notions of identification and authorization of all entities involved, defined the roles and purpose of the entities, as well as what information is needed for devices to be updated. The architecture handles the steps from deploying a new device to updating it and keeping it operational after updates. In addition a manifest format was proposed. The next section will present a life cycle perspective on device management for updates, augmenting the architecture with an understanding of how devices can remain functional for a long time.

3.2 Internet of Things Update Life Cycle

The previous section discussed five key areas that are all part of the life cycle of a device, which is presented in this section. From the moment a new device is deployed in a network to the point where it is taken out of service possibly several years later, the life cycle describes a holistic

Figure 3.6: The life cycle of a device.



view of the state and operations of devices.

Figure 3.6 summarizes the life cycle of a device from an update perspective. The figure shows the different stages of a device from being manufactured to ending its service. The figure also contains annotations showing what needs to be done in each stage. These stages are discussed in further detail in sections 3.2.1-3.2.3.

Note that the life cycle does not mention authorization. Authorization tokens are far too short lived compared to certificates and profiles to be of concern in the life cycle. Requesting and verifying tokens would happen in the maintenance stage when a device is being updated, but it is a level of detail too fine for the life cycle model.

3.2.1 Predetermined Information

A factory new device that is to be installed in an IoT network needs to be shipped with some information in order to enroll and register at a server and trust future communications. A pre-shared key and CA certificate are needed for the device to trust the CA and for the CA to know the device is supposed to receive a certificate. The device must be aware of its own vendor and class IDs in order to verify manifests and register at an update server. Registration also requires a known registration endpoint, this can be a well-known endpoint implemented by all update servers. Lastly, the device needs whitelists of operators and update servers in order to trust traffic originating from them. Any operator or update server that is not whitelisted is to be ignored.

3.2.2 Enrollment and Registering

With the required information a device can enter the enrollment stage. Devices obtain certificates by enrolling at the CA, which as previously mentioned requires a pre-shared key and CA certificate. After obtaining a certificate the device can be trusted by the servers and operators as well as other devices, allowing for secure communication.

The next stage is registering. By sending vendor and class IDs alongside its firmware version to a registration endpoint, devices can register and have an update server create a profile for them. The information in the profile tells through which protocols to reach the device and which version it is running. This information is transient and will change upon a successful update. Update servers must handle devices re-registering by either updating the relevant parts of the profile or creating an entirely new one. The approach chosen is implementation dependant.

3.2.3 Maintenance

After a device is enrolled and registered it enters the maintenance stage. The maintenance stage can be expected to last for several years and this is where the device performs its duties as well as receives and applies updates. After an update, a device will either move back to the enrollment or registration stage. If the certificate is broken after updating, as discussed in Section 3.1.2, the device moves back to the

enrollment stage to obtain a new certificate. After obtaining a new, valid certificate it can update its profile in the registration stage.

If the certificate is still valid after the update, there is no need to re-enroll and the device moves directly back to the registration stage, updating the profile at the servers. Finally the device moves to the maintenance stage once again, and so forth. The device will remain in the maintenance stage until it either breaks or is taken out of service. If you are a manufacturer of IoT devices please consider recycling or (securely) re-using devices, starting the life cycle anew.

3.2.4 Summary

This section has presented a life cycle view of a device in need of updates. Understanding the life cycle helps preparing the device for updates as well as letting it react accordingly to being updates. To continue being operational is a security concern and devices must be able to last for long, thus potentially applying many updates. With a proposed architecture and life cycle in place, the architecture can be profiled to provide implementers help when deciding upon an implementation. Profiles are the topic of the next section.

3.3 Internet of Things Update Architecture Profiles

The previous two sections have proposed a technology agnostic update architecture and a life cycle perspective for devices. It is time to show concrete incarnations of this architecture using state of the art IoT protocols. This section is aimed towards implementers needing to make decisions about how to implement the update architecture. This section will discuss choice of image digest algorithm, vendor and class ID generation, payload encoding and encryption, and considerations when choosing between DTLS/CoAP and OSCORE for an update architecture profile.

3.3.1 DTLS Profile

When implementing the architecture, one choice of protocols is using DTLS and CoAP for constrained communication to and from devices.

For enrollment and authorization, EST and ACE can be used with their respective suitable profiles. However, as DTLS provides security at the transport layer it cannot be used for broadcasting, OSCORE is an option better suited for broadcasting. Note that the architecture itself is broadcast friendly as it does not make security assumptions that would break broadcasting, but choice of protocols will influence the viability of broadcasting.

DTLS is already profiled for use in IoT contexts and EST and ACE profiles are works in progress [17], [31], [32]. The update architecture needs little extra profiling in addition to these protocols, namely image digest algorithm, vendor and class ID generation, payload encoding, and payload encryption. The security of the architecture is not bound to characteristics of any of these protocols and others can be used instead, this is just one possible profile.

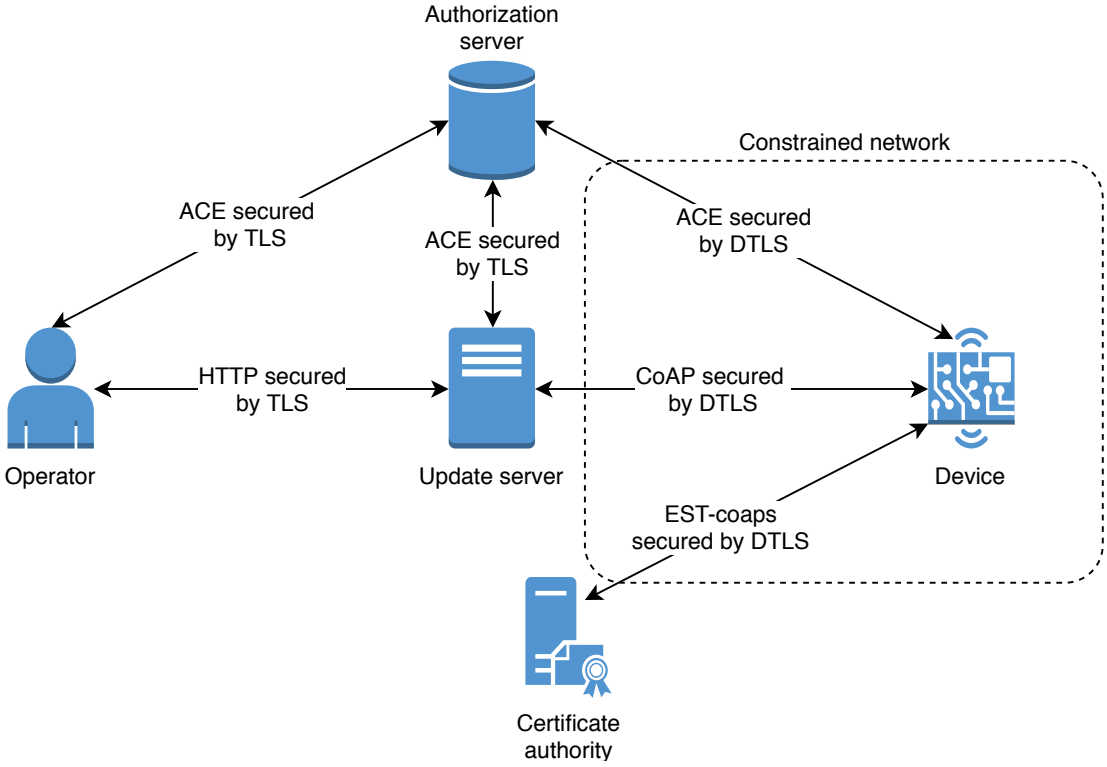
Figure 3.7 shows where in the constrained part of architecture the different protocols of the DTLS profile will be used. All communication to and from the device is reliant on DTLS, the only exception is the enrollment procedure which will require a pre-shared key as discussed in Section 3.1.2. The communication protocols in the non-constrained part of the architecture are suggestions and not part of the profile.

ID Generation and Hash Algorithms

The DTLS IoT profile suggests ciphersuites for the use of pre-shared keys and certificates and EST has been profiled with CoAPs to use the same ciphersuites for certificates. These suites implement the **SHA-256** hash function which can be used to calculate the digest of an image. Since the digest is just supposed to verify the integrity of the image, SHA-256 is an adequate choice and does not require implementing new hash algorithms for devices already running DTLS.

Generating vendor and class IDs should be done so that devices get unique identifiers and devices with the same names from different vendors do not clash. For this purpose, **UUIDs** can be used [33]. Version 3 and 5 of UUID are "name-based" versions meaning they operate in a given namespace. By using a registered domain name of the vendor, which is already guaranteed to be unique, as namespace a unique UUID can be generated for that vendor. By using the vendor UUID as the namespace for the class IDs, devices from different vendors can share names but still have different class IDs. A hierarchical namespace

Figure 3.7: The various protocols used in the DTLS profile. Protocols outside the constrained networks are suggestions.



structure is suggested also by SUIT.

Section 4.3 of the UUID specification states that "Choose either MD5 or SHA-1 as the hash algorithm; If backward compatibility is not an issue, SHA-1 is preferred". This means **UUID5** should be used as it uses SHA-1 for hashing. ID generation will not occur on the devices themselves, devices only need to be prepared with their IDs for comparing with the manifest.

Payload Encoding and Encryption

For encoding, **Concise Binary Object Representation (CBOR)** is an efficient binary encoding [34]. CBOR aims to be an extensible data format providing very small code sizes. It supports simple values as well as arrays and maps meaning it is easy to map to and from JSON while being more compact than JSON.

The use of CBOR enables the use of **CBOR Object Signing and Encryption (COSE)** [35]. COSE provides encryption and signing for

CBOR encoded objects, which in the architecture will be the manifest and image. As DTLS secures the channel, the manifest and image can be signed using COSE objects to ensure their integrity during transport.

3.3.2 OSCORE Profile

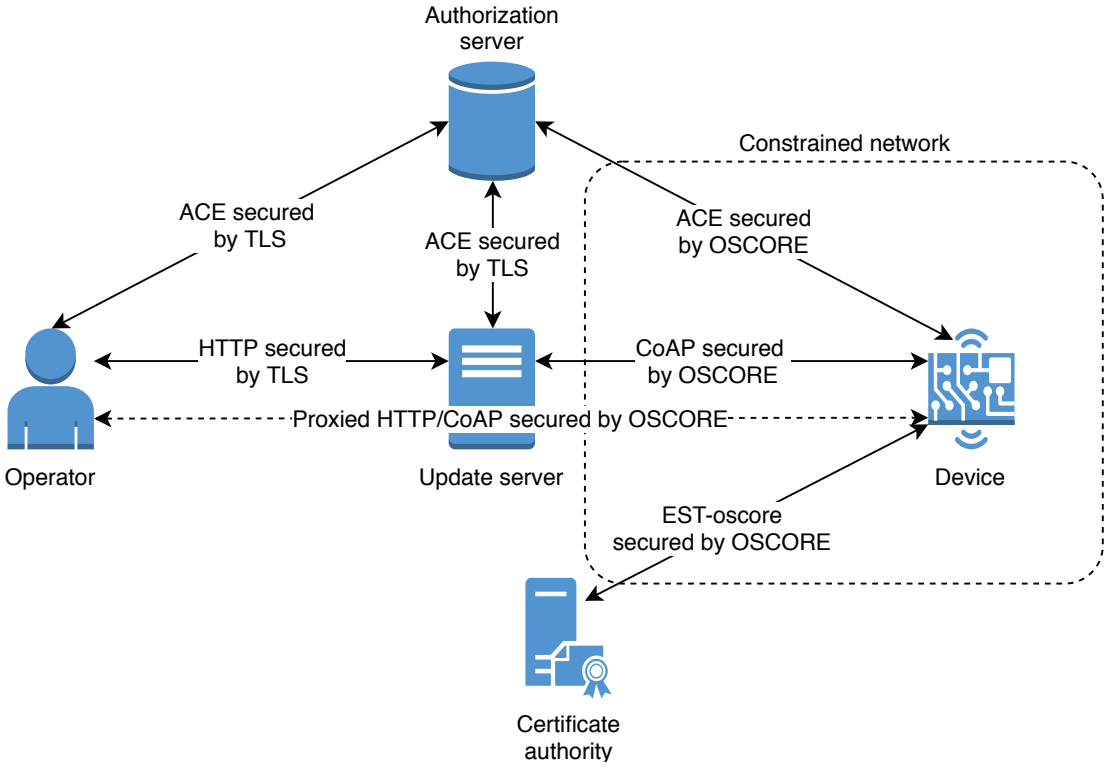
Object Security for Constrained RESTful Environments (OSCORE) provides end-to-end protection for CoAP messages using COSE [16]. By providing end-to-end encryption CoAP message security is not terminated at a proxy or update server as with DTLS. If updates are being pushed from an operator to a device directly, OSCORE can support HTTP/CoAP proxying to secure the entire transport. The update server will not be able to view or manipulate the protected contents. As update servers are in addition to transporting updates supposed to act as update repositories, a decision has to be made how updates are stored on the server. This is an implementation detail.

Figure 3.8 shows where in the constrained part of architecture the different protocols of the OSCORE profile will be used. All communication to and from the device is reliant on OSCORE, the only exception is the enrollment procedure which will require a pre-shared key as discussed in Section 3.1.2. The communication protocols in the non-constrained part of the architecture are suggestions and not part of the profile.

OSCORE can run directly on top of UDP and supports broadcasting as well as unicast. In the case of broadcasting, a security context must be defined [36]. If a security context is in place, unicast OSCORE is still possible. At the time of writing, there are two Internet-Drafts aiming to specify profiles for ACE and EST using OSCORE, meaning these protocols can be used for enrollment and authorization with OSCORE as well [19], [37]. ID generation, hash algorithms, and payload encoding and signing in the OSCORE profile is the same as in the DTLS profile, see Section 3.3.1.

When trying to achieve end-to-end encryption key management becomes more difficult, how do you ensure end-to-end encryption will still allowing all intended recipients (devices) to decrypt the payload? One solution is to let the operator encrypt the payload with a symmetric key which the update server then either encrypts with each intended recipients asymmetric key or with a group key. This avoids having to re-encrypt the payload (breaking end-to-end encryption) and allows

Figure 3.8: The various protocols used in the OSCORE profile. Protocols outside the constrained networks are suggestions.



storing the encrypted payload and just re-encrypting the key for new recipients. This approach means a trusted communication must be established between the operator and update server for securely exchanging the symmetric key of the payload and that update servers must be authorized to retrieve said key.

Another interpretation of end-to-end encryption could be purely between the update server and devices. If a constrained network possesses some routing capabilities an update server might not know which intermediaries will be used for transporting a package. In this case, OSCORE can be used to achieve end-to-end security between the update server and the intended device, while leaving operator-to-update server communication to some other security mechanism. This is highly topology dependant.

While being better suited for broadcasting and providing end-to-end security, OSCORE introduces some key management issues and is not yet standardized. The specification ([16]) is a work in progress as is many of the profiles mentioned in this section. DTLS and its related IoT profiles are standardized and much more mature. If implementing the architecture using OSCORE, it is worth noting the incomplete status of the standard.

3.3.3 Summary

This section identified two options for implementing the architecture: either CoAP over DTLS or OSCORE. DTLS does not support broadcasting however which might be of importance, for this purpose OSCORE can be used instead. OSCORE does not rely on a secure channel established by DTLS but instead provides end-to-end security for CoAP messages using COSE. OSCORE is, unlike DTLS, not yet standardized and there is a partial implementation of OSCORE featuring only COSE encryption objects in a Contiki fork [38]. Profiles surrounding DTLS for EST-coaps and ACE are also closer to standardization than their OSCORE counterparts.

Both profiles use SHA-256 for calculating image digests, UUID5 for vendor and class IDs using registered vendor domains, EST with its corresponding profile for enrollment, and ACE with its corresponding profile for authorization. Payloads are encoded and signed using CBOR and COSE for both approaches, ensuring integrity.

With the proposed profiles, a prototype can be implemented and

evaluated. Implementation of a prototype is the topic of the next section.

3.4 Implementation

The previous sections proposed the update architecture of the thesis, its key components, and security considerations such as identity and access control. This section will discuss a prototype implementation of the architecture as well as a manifest generator.

3.4.1 Manifest Implementation

As manifests contain certain information difficult for humans to provide such as monotonically increasing sequence numbers and hash digests, a manifest generator was created to help test the prototype [39]. It is a Python script which accepts information about vendor and class namespace, version, image file, and associated URL in order to generate and format a manifest both in JSON and CBOR. The outputted manifest follows the format specified in Section 3.1.6 and features all required fields although some left blank. It is a bare-bones manifest containing only the required information for a singular, monolithic update. There are no dependencies or options specified.

The manifest is a JSON map featuring the required elements shown in Figure 3.2. The elements are in the same order as in the figure but the names of the fields have been substituted for integers in order to save space. Table 3.1 shows this mapping. Repeatable structures such as preconditions are described as a nested array of maps, where each map in the array constitutes one instance of such an element. The keys in these nested maps are also mapped to integers as in the main manifest structure, but resetting the counter for each map. The mapping of keys in nested structures is shown in Table 3.2. The example manifest used can be found in Appendix A.

Table 3.1: Mapping manifest elements to integers as keys in the JSON manifest.

Element Name	Mapped To
versionID	0

Table 3.1: Mapping manifest elements to integers as keys in the JSON manifest.

Element Name	Mapped To
sequenceNumber	1
preConditions	2
postConditions	3
contentKeyMethod	4
payloadInfo	5
precursorImage	6
dependencies	7
options	8

Table 3.2: Mapping elements in nested structures to integers.

Element Name (corresponding structure)	Mapped To
type (conditions and options)	0
value (conditions and options)	1
URL (URL/digest pair)	0
digest (URL/digest pair)	1
format (payload info)	0
size (payload info)	1
storage (payload info)	2
URL/digest pair (payload info)	3

In the client code, the manifest is received as a string, still formatted as JSON. In order to parse it, structs resembling the format of the manifest were created, see Listing 3.1. The base manifest structure contains values of version ID, sequence number, and content key method alongside pointers to other parts of the manifest. These parts, represented as repeated maps in the JSON string, are implemented as linked lists in order to store an arbitrary amount of such structures. With the example manifest shown in Appendix A, the base manifest structure would for instance contain a two element long linked list of preconditions (vendor ID then class ID). Certain manifest elements, namely precursor image, dependencies, and URL/digest pair in payload info convey the same

kind of information but are separated into different lists because of the differences in semantics.

```
1   typedef struct manifest_s {
2       uint8_t versionID;
3       uint32_t sequenceNumber;
4       struct condition_s *preConditions;
5       struct condition_s *postConditions;
6       uint8_t contentKeyMethod;
7       struct payloadInfo_s *payloadInfo;
8       struct URLDigest_s *precursorImage;
9       struct URLDigest_s *dependencies;
10      struct option_s *options;
11  } manifest_t;
12
13  typedef struct condition_s {
14      int8_t type;
15      char *value;
16      struct condition_s *next;
17  } condition_t;
18
19  typedef struct payloadInfo_s {
20      uint8_t format;
21      uint32_t size;
22      uint8_t storage;
23      struct URLDigest_s *URLDigest;
24  } payloadInfo_t;
25
26  typedef struct URLDigest_s {
27      char *URL;
28      char *digest;
29      struct URLDigest_s *next;
30  } URLDigest_t;
31
32  typedef struct option_s {
33      int8_t type;
34      char *value;
35      struct option_s *next;
36  } option_t;
```

Listing 3.1: The client manifest implementation

3.4.2 Prototype Implementation

The prototype used in the thesis is developed in order to measure the efficiency of transport during an update procedure as well as serving as a source of inspiration for other implementers. It consists of a server and a client both implemented in Contiki-NG, thus written purely in C. The server is implemented in Contiki-NG as a proof-of-concept that a more capable IoT device could be used as an update server. The prototype uses a pull model meaning the client initiates the update procedure and the server responds with a corresponding resource. All traffic is sent via CoAPs, encrypted by DTLS. Certificate support in Contiki-NG was at the time of implementation missing, thus pre-shared keys were used instead.

The server implements three resources mapped to the endpoints update/register, update/manifest, and update/image. The register resource listens to POST requests and upon a request extracts the vendor id, class id, and version number sent by the client and creates a profile file before answering with message code 2.01 CREATED. Other information can be put into the profile, such as choice of protocol and IP address, in order to reach the device again. For the prototype, such information is not needed, and the file is created just as proof-of-concept.

In a real deployment with different devices, thus different manifests, the manifest resource is responsible for picking a suitable manifest based on the information in the device's profile. For the prototype only one manifest is used which is hard-coded into the manifest resource. Upon a GET request the manifest is encrypted through COSE and sent using CoAP's block option. The entire manifest is encrypted at once and the ciphertext sent block by block. Since the manifest is relatively small it is easy to allocate buffers large enough to encrypt the entire manifest at once, leading to smaller overhead. Ideally the manifest would be signed with COSE instead of encrypted, but COSE implementations in Contiki-NG are limited, at the time of development signing was not available and implementing it would prove too time consuming. Encrypting the manifest requires a bit more information than signing, such as a nonce and Additional Authenticated Data. These are hard coded in both client and server, alongside the key.

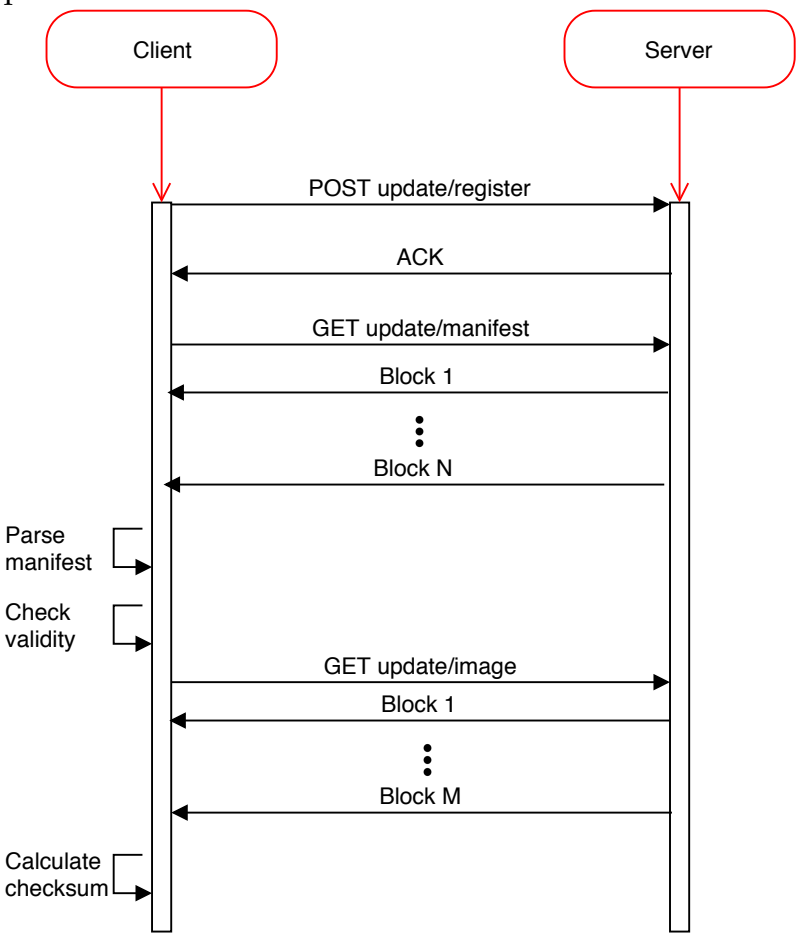
Lastly, the image resource generates data in a deterministic manner, COSE encrypts it block by block, and sends through CoAP's block option. Ideally this data would also be signed instead of encrypted.

The reasoning for encrypting the data block by block is that should a large amount of data be sent, such as a firmware image, it would be difficult to allocate buffers large enough to process the entire image at once, both for the server and client. Instead it is encrypted and decrypted blockwise. While this solves the issue of encrypting and decrypting large amounts of data, it introduces a larger overhead as each 32 byte block sent will contain 24 bytes of data and 8 bytes of tag for validation. The manifest resource only introduces this overhead once while the image resource introduces it once per block.

The client performs three remote calls to the server, one call to each resource, and performs some additional operations related to the manifest and image data. First a POST request is sent to the update/register endpoint. The client does not need to act upon the response. Afterwards a GET request is sent to the update/manifest endpoint. The manifest callback buffers the manifest into memory block by block and when the callback finishes decrypts the entire manifest at once. After decryption it is parsed into structs shown in the previous section and pre-conditions are checked. If any other fields that need checking before proceeding, such as optional pre-directives, they are also checked now.

If the client deems the manifest to be correct it uses the URL found in the payloadInfo->URLDigest struct to call upon that endpoint. In this case it is update/image, but could be any endpoint hosting a resource specific for that class of device. The image callback decrypts each block and updates a SHA-256 context with the plaintext data. After transfer of all image data, the checksum calculation is carried out to generate a checksum of the image data. This concludes the transfer of the update. Figure 3.9 summarizes the interactions between client and server.

Figure 3.9: The interactions of client and server during an update procedure.



Chapter 4

Results

The previous chapter introduced the update architecture proposed by the thesis and the design of a prototype aimed to evaluate it. This chapter will explain how the evaluation was carried out, its results, as well as analyze the proposed architecture qualitatively from the SUIT specification.

4.1 Quantitative Evaluation of Update Architecture

This section present the results of a experiment carried out to measure the energy consumption of the prototype on Firefly boards. The Firefly board features a ARM Cortex-M3 with 512 KB flash and 32 KB RAM and supports IEEE 802.15.4 communication [40]. In addition to energy consumption, code size and communication overhead is also reported.

4.1.1 Energy Consumption

Contiki-NG provides a utility called Energest which can measure how much time, measured in ticks, is spent by a program. It is divided into five categories: CPU, Low Power Mode, Deep Low Power Mode, Transmit, and Listen. CPU, Low Power Mode, and Deep Low Power Mode are related to the work of the processor while Transmit and Listen measures the radio. Each type can be measured independently by calling the function `energest_type_time(energest_type_t type)` where type is the category you want to measure. For the experiment, the client measures the register callback, the manifest callback,

the decoding and parsing of the manifest, and the image callback. The server measures the register resource, the manifest resource, and the image resource. The entire block transmission of the manifest and image resources are measured, and both client and server measures all five categories. How measurements are taken is shown in Listing 4.1.

```
1     energest_flush();   // To update values
2     t1 = energest_type_time(ENERGEST_TYPE_TO_MEASURE);
3     // Do operations
4     energest_flush();   // To update values
5     t2 = energest_type_time(ENERGEST_TYPE_TO_MEASURE) - t1;
```

Listing 4.1: How to measure ticks in energest.

With the amount of ticks, the following formula can be used to obtain energy consumption (measured in Joule)

$$Energy(J) = \frac{ticks}{ticks/second} * V * A$$

where ticks per second is defined in the platform dependant `ENERGEST_SECOND` macro. When gathering data about energy consumption, the client and server were flashed onto a Firefly board each and connected to a host laptop through USB. All communication between client and server was wireless and data was printed out to a terminal via serial port. The amount of blocks sent by the image resource varied between 500, 2000, and 4000 with ten runs per block size. The registration and manifest related operations were held constant as the client always registers with the same information and the server sends the same manifest.

Figures 4.1 and 4.2 shows average energy consumption for registration and manifest related operations for both client and server over the thirty runs. Figures 4.3 and 4.4 shows average energy consumption for the client and server when encrypting, transferring, and decrypting image data per block size, ten runs per block size.

Figure 4.1: Average energy consumption for client operations.

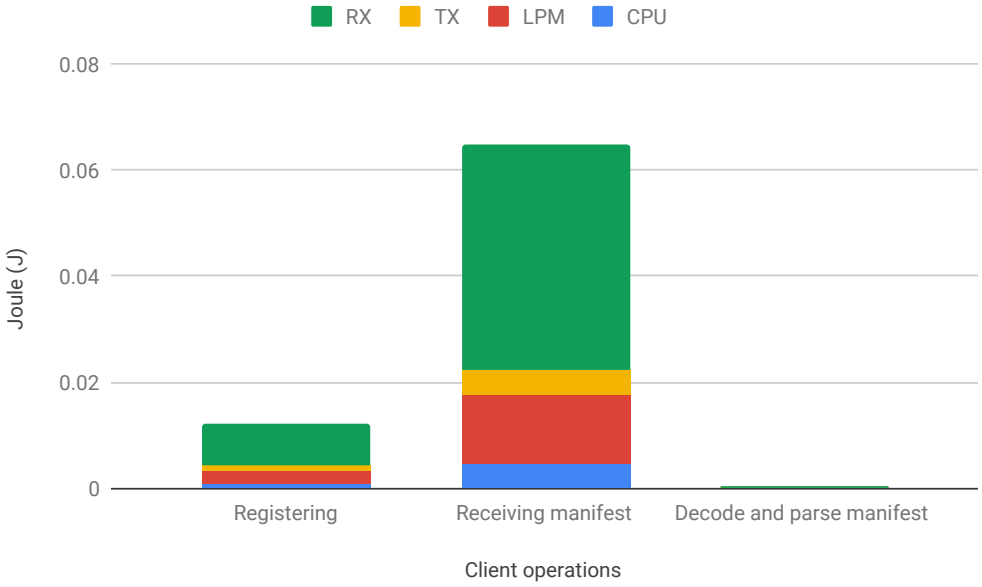


Figure 4.2: Average energy consumption for server operations.

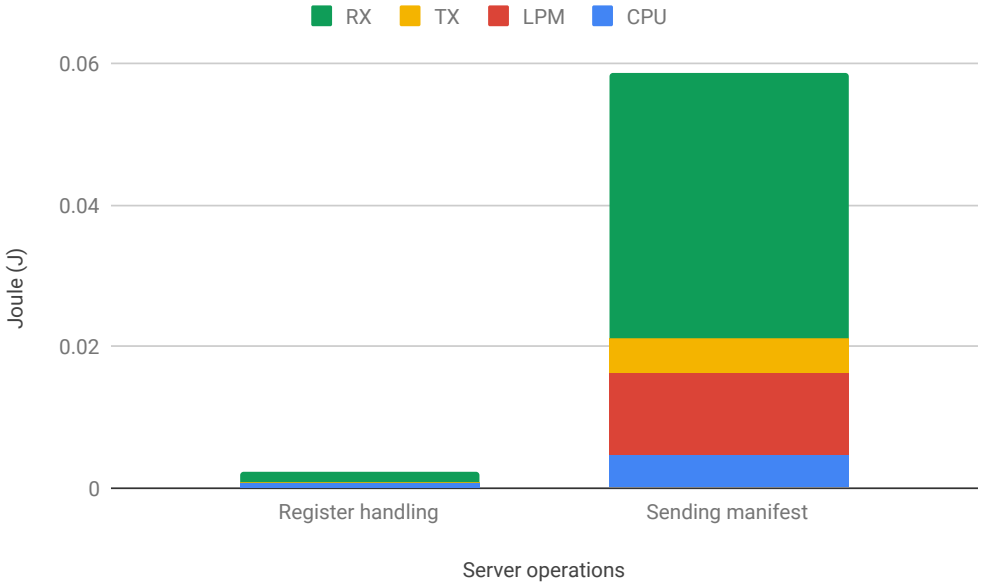


Figure 4.3: Average energy consumption for client during image transfer.

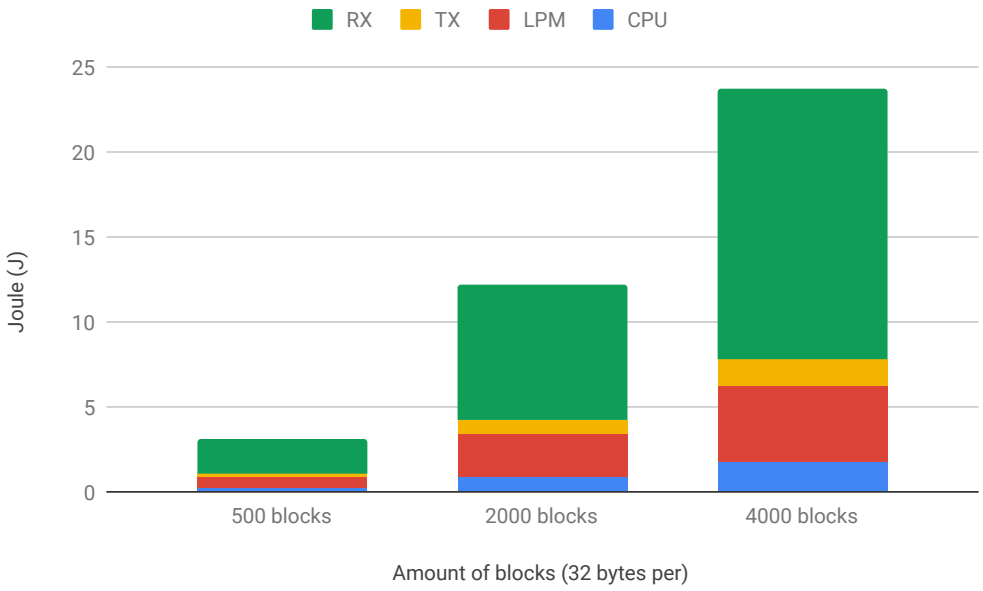
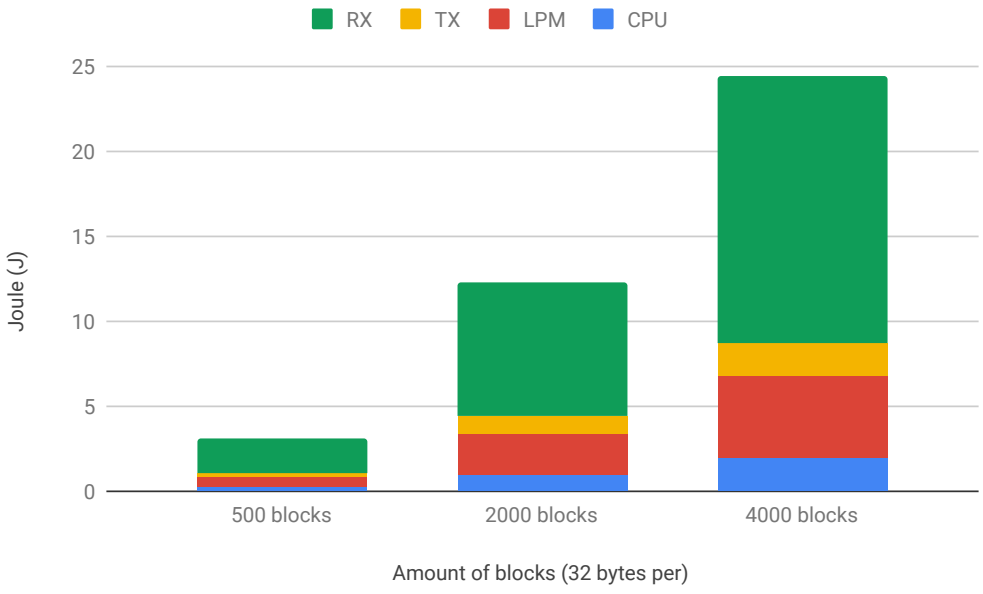


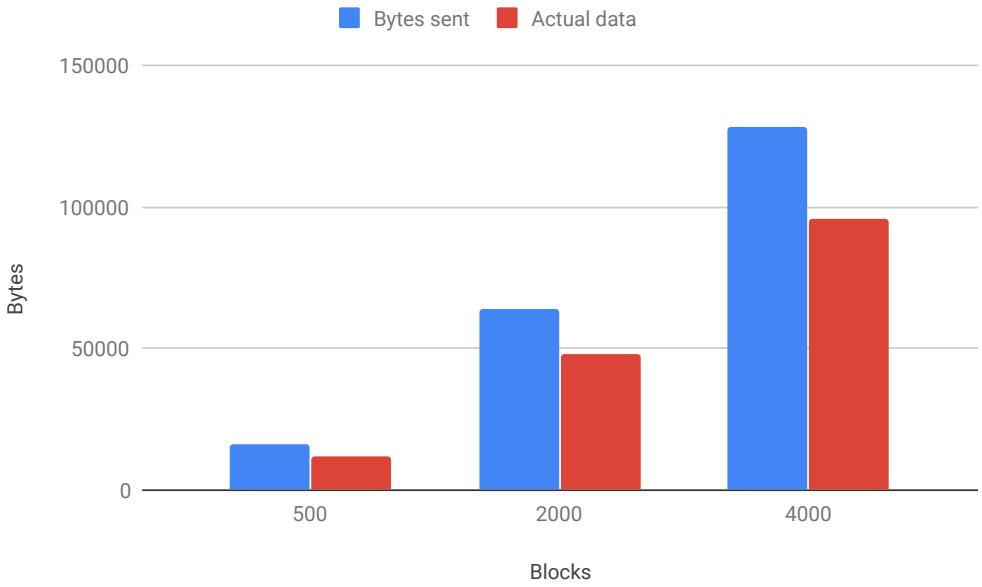
Figure 4.4: Average energy consumption for server during image transfer.



4.1.2 Communication Overhead

As discussed in Section 3.4.2, some overhead is introduced at the application layer as a consequence of using COSE encrypt. Each CoAP block sent by the server during image transfer contains 32 bytes of data of which 8 is a tag used to validate the ciphertext. Figure 4.5 shows this disparity.

Figure 4.5: Bytes transfered vs actual data sent measured in bytes at the application layer.



4.1.3 Code Size

When measuring code size the code was compiled with all debug macros set to 0, the energest measurements removed, and no other Contiki-NG modules except CoAP. As a point of reference, a bare-bones example was created. The bare-bones example only contains the necessary imports to Contiki-NG and its CoAP engine and declares a process with no code in it. It is the smallest runnable example for Firefly devices using the same project configuration as the other source files. Its source code is found in Appendix C. By running the tool arm-none-eabi-size on the ELF-files the code size can be obtained, which is shown in Table 4.1.

text	data	bss	dec	hex	filename
69982	1772	11303	83057	14472	bare-bones.elf
75956	1788	13599	91343	164cf	update-client.elf
77924	1884	11835	91643	165fb	update-server.elf

Table 4.1: Code size for client and server prototypes.

4.2 Qualitative Evaluation of Update Architecture

The proposed architecture can be evaluated in two parts analogous to how the SUIT working group presented their work, namely architecture and information model. Using the requirements of each document, a qualitative evaluation of the proposed architecture can be performed. The requirements are gathered from [21], [22].

4.2.1 Architecture

The SUIT architecture document specified ten requirements a suitable update architecture should fulfill:

Agnostic to how firmware images are distributed: No parts of the architecture assumes a specific suite of protocols or algorithms to ensure secure delivery of updates. The architecture does specify the usage of technologies such as certificates, asymmetric encryption, and authorization tokens, but these elements can be implemented in a wide variety of ways using different protocols, algorithms, and certificate/token types. The profiles provided in the thesis are examples of how specific incarnations of the architecture could look like.

Friendly to broadcast delivery: The architecture itself does not limit broadcast delivery and through correct usage of the manifest broadcasted updates will not be incorrectly installed by devices other than the intended recipients. Choice of technology can limit the usage of broadcasting, such as DTLS, but this is implementation specific.

Use state-of-the-art security mechanisms: The architecture is based on asymmetric cryptography using certificates as well as access

authorization through whitelists and possibly tokens for fine-grain authorization. Choice of key algorithms and token types will affect the cryptographic strength of the system, allowing for both smaller and less strong keys and tokens or larger and strong keys and tokens.

Rollback attacks must be prevented: By specifying a monotonically increasing sequence number in the manifest, devices can make sure they are installing fresh images. Manifests are signed through COSE which ensures authenticity, an assailant would not be able to modify a manifest and then recompute a valid signature.

High reliability: This is an implementation specific requirement, however the storage element of the manifest aids in achieving safe storage of a new image. After a successful update, devices are to re-register at servers. No acknowledgment means the server knows the update still must be applied, thus an interrupted update can be redistributed.

Operate with a small bootloader: The thesis suggests to store an unencrypted image alongside its digest for the bootloader to be minimal, only needing support for calculating a SHA256 digest of the image at boot. All information about whether or not to perform the update is encoded in conditions in the manifest, and can be stored with small memory usage.

Small parsers: The manifest format used in the thesis is simple, complete, and extensible. Parsing it requires storing the key/value pairs in predefined structures which is possible with a very small parser.

Minimal impact on existing firmware formats: The architecture makes no assumptions about firmware formats. It is up to the implementer to ensure the images are prepared for updating and that the bootloader contains the necessary functionality. Transporting the images is the same regardless of content.

Robust permissions: The architecture is based on asymmetric cryptography using certificates for identity and confidentiality, and whitelists of servers and operators as well as tokens for authorization. Different deployments have different security and performance requirements, the architecture is flexible by allowing

different kinds of key algorithms, encryption schemes, and token types. Some deployments may not need tokens at all as authentication is done on a device level and whitelists are sufficient, white other deployments wanting to update specific pieces of a device may need fine-grained authorization through tokens. Different devices in the same deployment can have different authorization configurations, for instance by accepting different types of tokens.

Operating modes: The architecture supports the device initiated pull model as well as the operator initiated push model. The update server acts as a mediator between device and operator as well as a repository for images and device profiles, letting the operator query the server for device statuses and devices query for updates depending on the model used.

4.2.2 Information Model

The information model had more requirements posed upon it than the architecture and its implementation forms an important basis for carrying out safe updates. Evaluating the proposed manifest format and implementation against the requirements of the specification yields the following:

Installation instructions: Installation instructions or any sort of directive is not included in the base manifest structure but can be implemented in the options field. Directives, shown as an option in Figure 3.2, could be used to indicate decompression algorithms, preparation of bootloader etc while other more specific instructions can be implemented either as a directive or a new kind of option element specific to that deployment.

Override non-critical manifest elements: The proposed manifest format has the critical elements as a baseline and the rest severed into the options field. Defining new option types allows for new elements in the manifest, but the devices need to be updated for their parsers and manifest checkers to be aware of the new elements.

Modular update: Supported through precursor images, dependencies, URI aliases, and the use of authorization tokens for more granular

updates. For devices with several MCUs and applications, possibly several operating systems, update access can be controlled through authorization tokens such that a vendor can only update their respective part.

Multiple authorizations: Enforced via operator and update server whitelists and authorization tokens.

Multiple payload formats: The architecture and manifest format does not make assumptions about the payload formats. The mappings of formats to keys in the manifest is deployment specific and whatever formats a deployment might opt to use can be represented in the manifest together with a SHA256 digest of said format.

Prevent confidential information disclosure: The architecture is based on state-of-the-art security mechanisms and achieves confidentiality through asymmetric cryptography. Both manifest and payload are encrypted and then signed using COSE.

Prevent devices from unpacking unknown formats: Supported through the use of the manifest format element, preconditions, overridable/custom directives, and letting the update server decide if there is a suitable update for a device by matching against its registered profile.

Specify version numbers of target firmware: Since devices are required to register their version alongside vendor and class ID, operators can query device statutes from the update server and then prepare updates matching these categories of devices.

Enable devices to choose between images: Several images can be specified in a single manifest through the use of either precursors, dependencies, URIs (mirror list), or aliases depending on the specific use case. New options can also be defined capturing this behaviour, such as a new option providing URL/Digest pairs intended for parallel storage.

Secure boot using manifests: The thesis has not examined bootloaders, thus this requirement is out of scope. Investigating secure bootloaders could be part of future work in the topic.

Decompress on load: This behaviour can be encoded in the payload format of the manifest, or through the use of a custom directive indicating a compressed payload is stored.

Payload in manifest: Can be added as the optional "payload" element, the value would be the data of the payload and the regular payloadInfo element would contain its digest. Worth noting is that for most deployments including the payload in the manifest will dramatically increase the size of manifest, thus a manifest parser must be prepared for it.

Simple parsing: As explained in the previous section, the manifest format is simple and easy to parse even for constrained devices.

Chapter 5

Discussion

5.1 Conclusions

The thesis has presented a software and firmware update architecture for IoT. The architecture is based on requirements developed by the IETF SUIT working group and aims to be interoperable and usable for very constrained devices and state-of-the art security mechanisms such as asymmetric cryptography, certificates, and authorization tokens. The architecture defines the roles of devices, update servers, and operators, discusses how these actors can authorize themselves and what tokens are needed, the operations of these actors such as registering devices, and briefly considers local upgrade handling. Furthermore the thesis discusses the device life cycle for new devices entering a network, being updated, and continuing their service for many years, where devices must re-register and possibly re-enroll following certain operations of an update procedure. Two different profiles for the architecture, one using DTLS and CoAP and the other OSCORE, were presented and the DTLS/CoAP profile was implemented in a prototype with some restrictions.

Measuring code size of the update client and server as well as a bare-bones Contiki-NG example, it is evident that the solution can be applied to very constrained systems. The difference in size between the client and bare-bones example is 8286 bytes and the server and bare-bones example is 8586 bytes. In a real deployment the code size will increase as the prototype's manifest parser is rudimentary and the client does not contain any code for preparing the bootloader, but applying the architecture to constrained devices is still feasible.

Experiments on energy consumption found that transfering the image data makes up the vast majority of energy consumed during an update procedure. This is due to the register operations and sending/receiving of a manifest sends a lot less data. Also when the client parses the manifest the operations are entirely local to the client, not using any expensive radio operations. During image transfer the largest source of energy consumption is the radio in receive mode followed by the CPU in Low Power Mode. The amount of time thus energy needed for the transfer scales linearly with the size of the data being sent, meaning opportunities to send less data are of importance when it comes to saving energy.

The architecture introduces some communication overhead on the application layer when sending both manifest and image data. When encrypting with COSE a 8-byte tag is added for validation to the ciphertext. In the case of the manifest this tag is added once as the manifest is small enough to be encrypted at once. For the much larger image data, which cannot be encrypted all at once, each block is encrypted individually instead. This means that each block introduces 8 bytes of tag for each 32 bytes of data sent in a CoAP block. By sending less data a smaller amount of blocks must be transfered meaning less overhead is sent in total. This furthers the argument for exploring opportunities that reduce the amount of data sent, such as differential updates.

5.2 Security Considerations

Despite being able to add timestamps in the manifest specifying a time of update it can prove difficult to install updates at the right moment. Timestamps are also unable to account for if processes are being interrupted or not, one might want to postpone updates until a certain process or thread is idle. Not having fine-grained controlled over when an update is installed is especially important for devices used in safety critical contexts such as in factories

The architecture does not support scenarios where redundant devices are used to achieve the same goal, such as monitoring equipment. To update one of the devices while the other operates and then update the other would require manual intervention such as sending the devices updates one at a time. This is a rather brittle approach and having support for twinned devices would help safety critical contexts.

The architecture does not account for physical threats, only threats that arise during wireless transport. The discussion in Section 3.1.5 does not account for scenarios where the physical storage of devices is compromised. Implementers are advised to store manifest and image data in a way that makes sense for their own deployment, whether that is unencrypted, encrypted, or encrypted manifest with unencrypted image.

5.3 Future Work

Moving forward, research on preparing bootloaders for upgrades and safe boot as well as performing differential updates would be of importance. As the results showed, the largest source of energy consumption is the radio receiver during transfer of updates, largely because of the relatively large size of firmware images. By performing differential updates instead the size of images can be reduced and time to retrieve them and thus energy consumption will go down. Furthermore, developing a prototype using certificates instead of pre-shared DTLS keys, authorization tokens, and COSE signing instead of encryption would further aid the understanding of the architecture's requirements on devices and how feasible it is to use for less constrained devices conscerning code size and energy consumption.

Bibliography

[1] Peter Jonsson, Stephen Carson, Per Lindberg, et al. *Ericsson mobility report*. Nov. 2018. URL: https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf.

[2] Nicole Perlroth. *Hackers Used New Weapons to Disrupt Major Websites Across U.S.* Oct. 2016. URL: https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html (visited on 01/16/2019).

[3] Alex Hern. *Hacking risk leads to recall of 500,000 pacemakers due to patient death fears*. Aug. 2017. URL: https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update (visited on 01/16/2019).

[4] Qi Jing, Athanasios V Vasilakos, Jiafu Wan, et al. "Security of the Internet of Things: perspectives and challenges". In: *Wireless Networks* 20.8 (2014), pp. 2481–2501.

[5] Md Mahmud Hossain, Maziar Fotouhi, and Ragib Hasan. "Towards an analysis of security issues, challenges, and open problems in the internet of things". In: *2015 IEEE World Congress on Services*. IEEE. 2015, pp. 21–28.

[6] Arwa Alrawais, Abdulrahman Alhothaily, Chunqiang Hu, et al. "Fog computing for the internet of things: Security and privacy issues". In: *IEEE Internet Computing* 21.2 (2017), pp. 34–42.

[7] Syed Ali, Ann Bosche, and Frank Ford. *Cybersecurity Is the Key to Unlocking Demand in the Internet of Things*. Oct. 2018. URL: https://www.bain.com/insights/cybersecurity-is-the-key-to-unlocking-demand-in-the-internet-of-things/ (visited on 12/10/2018).

[8] *Software Updates for Internet of Things (suit)*. URL: https://datatracker.ietf.org/wg/suit/about/ (visited on 01/16/2019).

[9] Stefano Babic. *swupdate*. 2019. URL: https://github.com/sbabic/swupdate (visited on 04/29/2019).

[10] Mender Software. *mender*. 2019. URL: https://github.com/mendersoftware/mender (visited on 04/29/2019).

[11] JUUL Labs OSS. *mcuboot*. 2019. URL: https://github.com/JuulLabs-OSS/mcuboot (visited on 04/29/2019).

[12] Jon Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[13] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347.

[14] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252.

[15] Carsten Bormann and Zach Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016. DOI: 10.17487/RFC7959.

[16] Göran Selander, John Mattsson, Francesca Palombini, et al. *Object Security for Constrained RESTful Environments (OSCORE)*. Internet-Draft draft-ietf-core-object-security-15. Work in Progress. Internet Engineering Task Force, Aug. 2018. 84 pp.

[17] Peter Van der Stok, Panos Kampanakis, Michael Richardson, et al. *EST over secure CoAP (EST-coaps)*. Internet-Draft draft-ietf-ace-coap-est-08. Work in Progress. Internet Engineering Task Force, Feb. 2019. 49 pp.

[18] Max Pritikin, Peter E. Yee, and Dan Harkins. *Enrollment over Secure Transport*. RFC 7030. Oct. 2013. DOI: 10.17487/RFC7030.

[19] Göran Selander, Shahid Raza, Martin Furuhed, et al. *Protecting EST payloads with OSCORE*. Internet-Draft draft-selander-ace-coap-est-oscore-01. Work in Progress. Internet Engineering Task Force, Sept. 2018. 9 pp.

[20] Ludwig Seitz, Göran Selander, Erik Wahlstroem, et al. *Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)*. Internet-Draft draft-ietf-ace-oauth-authz-22. Work in Progress. Internet Engineering Task Force, Mar. 2019. 81 pp.

[21] Brendan Moran, Milosch Meriac, Hannes Tschofenig, et al. *A Firmware Update Architecture for Internet of Things Devices*. Internet-Draft draft-ietf-suit-architecture-02. Work in Progress. Internet Engineering Task Force, Jan. 2019. 22 pp.

[22] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *Firmware Updates for Internet of Things Devices - An Information Model for Manifests*. Internet-Draft draft-ietf-suit-information-model-02. Work in Progress. Internet Engineering Task Force, Jan. 2019. 32 pp.

[23] Microsoft. *The STRIDE Threat Model*. Nov. 2009. URL: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20) (visited on 01/17/2019).

[24] contiki-ng. *Contiki-NG*. 2019. URL: https://github.com/contiki-ng/contiki-ng (visited on 01/18/2019).

[25] contiki-os. *Contiki*. 2018. URL: https://github.com/contiki-os/contiki (visited on 01/18/2019).

[26] "IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)". In: *IEEE Std 802.15.4-2003* (2003). DOI: 10.1109/IEEESTD.2003.94389.

[27] M. Kovatsch, S. Duquennoy, and A. Dunkels. "A Low-Power CoAP for Contiki". In: *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*. Oct. 2011, pp. 855–860. DOI: 10.1109/MASS.2011.100.

[28] contiki-ng. *TinyDTLS*. 2018. URL: https://github.com/contiki-ng/tinydtls (visited on 01/18/2019).

[29] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, et al. "Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. Boulder, Colorado, USA: ACM, 2006, pp. 29–42. ISBN: 1-59593-343-3. DOI: 10.1145/1182807.1182811. URL: http://doi.acm.org/10.1145/1182807.1182811.

[30] contiki-os. *Multithreading*. 2014. URL: https://github.com/contiki-os/contiki/wiki/Multithreading (visited on 01/23/2019).

[31] Hannes Tschofenig and Thomas Fossati. *Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things*. RFC 7925. July 2016. DOI: 10.17487/RFC7925.

[32] Stefanie Gerdes, Olaf Bergmann, Carsten Bormann, et al. *Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE)*. Internet-Draft draft-ietf-ace-dtls-authorize-05. Work in Progress. Internet Engineering Task Force, Oct. 2018. 20 pp.

[33] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. DOI: 10.17487/RFC4122.

[34] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Oct. 2013. DOI: 10.17487/RFC7049.

[35] Jim Schaad. *CBOR Object Signing and Encryption (COSE)*. RFC 8152. July 2017. DOI: 10.17487/RFC8152.

[36] Marco Tiloca, Göran Selander, Francesca Palombini, et al. *Group OSCORE - Secure Group Communication for CoAP*. Internet-Draft draft-ietf-core-oscore-groupcomm-03. Work in Progress. Internet Engineering Task Force, Oct. 2018. 32 pp.

[37] Francesca Palombini, Ludwig Seitz, Göran Selander, et al. *OSCORE profile of the Authentication and Authorization for Constrained Environments Framework*. Internet-Draft draft-ietf-ace-oscore-profile-07. Work in Progress. Internet Engineering Task Force, Feb. 2019. 26 pp.

[38] Martin Gunnarsson. *Contiki-NG*. 2019. URL: https://github.com/Gunzter/contiki-ng/tree/oscore_12 (visited on 03/26/2019).

[39] Simon Carlson. *manifest-generator*. 2019. URL: https://github.com/SimonCarlson/manifest-generator (visited on 04/05/2019).

[40] *Zolertia Firefly Revision A2 Internet of Things hardware development platform, for 2.4-GHz and 863-950MHz IEEE 802.15.4, 6LoWPAN and ZigBee® Applications*. ZOL-BO001-A2. Revision A2. Zolertia. Dec. 2017.

Appendix A

Example Manifest

The example manifest used during development and evaluation. It was generated by invoking `./manifest.py 500-blocks-manifest.json -i 500-blocks-data.txt -v test -c test -u update/image -m 1`. This command creates an output file named `500-blocks-manifest.json` using the data file `500-blocks-data.txt`, manifest version 1, and namespaces `test` for both vendor and class id. The data file contains the server generated data for 500 blocks.

```
{
    "0": 1,
    "1": 1556783337,
    "2": [{
        "0": 0,
        "1": "4be0643f-1d98-573b-97cd-ca98a65347dd"
    }, {
        "0": 1,
        "1": "18ce9adf-9d2e-57a3-9374-076282f3d95b"
    }],
    "3": [],
    "4": 0,
    "5": {
        "0": 3,
        "1": 11500,
        "2": 0,
        "3": [{
            "0": "update/image",
            "1": "37ce827871e6c63e9e5aab92f84
                ↪ c579f7aa5c6be0c3980cb6e30a102396abfd6"
        }
    ]},
```

```
22      "6": [],
23      "7": [],
24      "8": []
25  }
```

Listing A.1: The example manifest `500-blocks-manifest.json` used in the thesis, pretty-printed.

Appendix B

Repositories

Repository containing client and server update code, located in `examples/suitup`: `https://github.com/SimonCarlson/contiki-ng/tree/oscore_12`

Repository for manifest generation: `https://github.com/SimonCarlson/manifest-generator`

Appendix C

Bare Bones Example Source Code

```
1   #include "contiki.h"
2   #include "contiki-net.h"
3   #include "coap-engine.h"
4   #include "coap-blocking-api.h"
5   #include "coap-keystore-simple.h"
6
7   PROCESS(bare_bones_process, "Bare bones process");
8   AUTOSTART_PROCESSES(&bare_bones_process);
9
10  PROCESS_THREAD(bare_bones_process, ev, data) {
11  PROCESS_BEGIN();
12
13  PROCESS_END();
14  }
```

Listing C.1: Bare bones Contiki-NG example.