POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Libpull: The Portable Updates Lightweight Library

Securing Updates for Constrained Internet of Things Devices

Supervisor
Massimo Violante
Carlo Alberto Boano

Candidate
Antonio Langiu

March 2018

Summary

The Internet of Things (IoT) is radically changing the nature of the objects based on this technology, empowering them with the capabilities offered by the Internet, but also exposing them to a high number of security threats that could impact their security, safety, and the users' privacy. As every connected computer, also IoT devices need to be protected from software vulnerabilities by receiving software updates. This thesis focuses on the software update process for constrained IoT devices and explicitly targets Class 1 IoT devices, which are typically characterized by approximately 100 kB of ROM and 10 kB of RAM. The thesis contributes to mitigate the problem of software updates by designing and implementing *libpull*, a library exposing all the necessary functions to create an update systems for constrained IoT devices. The solution targets three main requirements: security, portability, and platform constraints. The architecture of the designed update system is based on two servers in order to protect the private key used for the main digital signature process, and also targets the problem of update freshness. The update is transmitted to the device using the CoAP protocol and an end-to-end encryption to grant confidentiality and authentication for both server and client. The digital signature is verified on the client using a Hardware Security Module (HSM), to safely store the keys and protect them from software attacks. The library architecture relies on many interfaces to increase its portability to different hardware platforms and operating systems. Moreover, libpull is protocol-agnostic, which enables the user to implement the network interface with the protocol that better fits the application requirements. The implemented solution does not explicitly target the activation phase, which can be performed by the user in many ways, such as static, dynamic or seamless software update, according to the constraints of the chosen hardware platform. The evaluation has been performed with two operating systems: Linux on the developing computer, and Contiki on a Class 1 IoT device (the TI CC2650 SensorTag). An experimental evaluation analyzed the memory footprint, the execution time and the energy consumption required to perform an update, by comparing different combinations of cryptographic libraries to perform the signature verification and network configurations to receive the update.

Ringraziamenti

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet of Things (IoT) introduces a new paradigm in human life. As suggested by the name, it poses its foundations on the Internet, the technology that changed the way we communicate and connect with people, empowering the process of exchanging information and ideas in the last three decades. The IoT will go beyond information and ideas only: including things from the physical world in the loop opens new ways to interact with the surrounding environment and the objects we use in our everyday life.

The IoT potentialities have been already seen in many application fields, such as smart buildings (by integrating smart objects into constructions to increase efficiency); home automation (by relying on smart objects to efficiently manage house parameters); smart cities (by collecting data from sensors to decrease pollution); smart manufacturing (by using smart sensors to reduce production costs and waste of raw material); smart healthcare (by tracking human body parameter and giving real-time assistance); As suggested by these examples, IoT is a fast-growing technology that finds application in quite every aspect of human life, and that will have a large impact in our society.

The number of connected IoT devices is expected to hit 26 billions by 2020 [1]. Since 2008, there are more objects connected to the Internet than people in the world. The reduction of the production costs will support this growth, allowing to include wireless radio modules even on cheap processors to offer remote control, monitoring, and sensing to practically every object.

Dissecting the IoT, we can find as smart object its core component. The term "smart" refers to the possibility of objects to become more interactive and aware. The advance of technology and the improved miniaturization process increases every day the number of objects that could become smart. These empowered devices enable new kind of interactions with humans and machines, using the Internet as the communication interface.

With the IoT transformation, a refrigerator could become smart by including food recognition capabilities and a lamp could become smart by changing its color according to the preferences of the user present in the room. The smartness integrated into these devices is usually accomplished incorporating a special purpose computer into them. If we consider the previous objects at the same level of computers, we could assert that a smart refrigerator is a computer able to keeps things cold and a smart lamp is a computer able to illuminate the room. This change of perspective helps us understanding this new technology deeply, focusing not only on what IoT could add to the objects, rather to what it could remove from the surrounding, which in many cases is privacy, security, and safety.

As every computer, smart objects are also affected by security vulnerabilities and need to be secured to protect their resources, such as collected data in case of sensors or physical interfaces in case of actuators. Not surprisingly, these devices were the target of many attacks in the last few years: some of them with the goal of collecting data, injecting malicious code, or access more worthy devices on the same network. Being IoT a relatively new technology, IoT devices are often not as secure as other systems based on more mature technologies, such as smartphones or computers, thus representing a good entry point to a local area network.

The reachability of smart objects from the public network makes them vulnerable to remote attackers, and their deployment in critical and private environments makes them a target for attackers aiming to exfiltrate information, hence affecting user privacy. The high number of poorly secured devices has been shown by Web services like Shodan [2], a search engine able to map the devices present on the Internet, enabling to search for them using particular keys or device classes, such as IP cameras or network printers. Moreover, many IoT devices must be deployed for many years, making them a good target for attackers looking for persistence.

Watching the problem from a more extensive angle, we can see the threats opened by billions of vulnerable IoT devices. In the second half of 2016, millions of poorly secured smart objects were part of an enormous Botnet called Mirai [3]. This network of compromised devices was mainly composed of smart cameras from a Chinese company shipped with default authentication credentials. This allowed the attacker to connect to them using the Telnet protocol and run arbitrary code, performing huge DDoS attacks able to take down critical services and infrastructures. After the release of the Mirai source code, many forks spread out targeting other vulnerabilities and involving other devices in the loop.

Considering the wide impact and benefits of IoT, security plays a critical role in the future of this technology. It can be seen as the basis to transform IoT into a reliable technology on which users, companies, and organizations can rely. Being IoT engaged in so many aspects of our society, its security will determine whether it will empower or harm our lives.

1.1 Problem Statement

The high number of attacks on IoT devices and their high success rate can be explained by the unpatched nature of smart objects. In fact, the actual perception of these devices does not make immediately evident their need of being secured. Users buying a smart lamp do not consider the need of keeping it protected from the last security vulnerabilities, as they would normally do with a standard computer. At the same time, lamp producers will face new challenges in protecting their products from threats they were not exposed before, requiring a high collaboration with security experts during the whole development phase. This makes even more clear the need to rethink the IoT paradigm, focusing on the needs of these devices when considering them to be at the same level of regular computers.

Last attacks on IoT devices were mainly abusing common vulnerabilities, known from decades on standard systems, such as hardcoded authentication credentials or buffer overflows. This indicates that the security problem of IoT is not merely technological, but instead economic. IoT devices are usually cheap, and the price imposed by the market does not give vendors the resources to integrate security features. Moreover, the technology evolution and the presence of many open source software and open hardware resources open the hardware market also to small companies, such as startups, often lacking the necessary resources and competencies to integrate security into their products. Furthermore, the low time to market of startups makes security one of the first aspects to be cut during the developing process.

The previous factors indicate the need for IoT to have reusable solutions able to decrease the costs and effort to integrate security in these devices. In particular, this thesis focuses on the security problem of software updates for IoT, a critical process still not included in many commercial systems. As shown by a survey analyzing the state of software updates in the embedded industry [4], 45,5 % of respondents said that they never integrated an update system into their solution. The other 54,5 % said that they developed their own update system in-house, confirming the identified need of having reusable solutions. The lack of an update system has been identified as one of the major threats in IoT security [5] and, considering the complexity integrated on some IoT software and that new vulnerability are discovered daily on OS and libraries, the need for a vendor of sending a software update to a device is just a matter of time.

The software update problem is not new in computer security and has been already analyzed in many other fields such as mobile, automotive, embedded systems. Even the security of a certain class of more powerful IoT devices has been investigated. This thesis focuses, instead, on a specific class of IoT devices: Class 1 constrained devices. Those are very limited in available memory and energy, and no generic solution is available today. Therefore, this thesis aims to solve this problem analyzing, designing and implementing an update system for constrained IoT devices, targeting security, portability, and the constraints imposed by these devices.

1.2 Requirements

Building an update system for constrained IoT devices is challenging, especially if the solution should to be reusable on many platforms to decrease the costs for the final IoT producers. Towards this goal, we identified three main requirements that should guide the design of the final solution: security, portability, and platform constraints.

- **Security.** Devices receive updates for two main reasons: integrate new features and fix security vulnerabilities. Especially for the second one, being able to ship updates becomes necessary for critical systems that could affect the safety of humans involved with them, such as applications in the home automation and healthcare fields. Moreover, the update process is a complicated process that requires high privileges, being generally in charge of replacing some modules or even the complete firmware of a device, and can hence quickly move from being a security feature to a security vulnerability. For this reason, it is essential to perform software updates ensuring integrity and authenticity of the received update, ensuring that an attacker is not able to inject code into a device exploiting the update system. Moreover, in many applications, the software running on the device represents a huge competitive value that needs to be preserved protecting the software confidentiality. For this reason, the update needs to be sent through an encrypted channel, granting confidentiality during the whole update process.
- **Portability.** IoT products rely on a multitude of devices and technologies, with different characteristics and requirements. This device diversity increases the challenges of building a single update system usable on many constrained IoT devices, since such an update system typically needs to deal with very device-specific details. Moreover, operating systems for constrained devices provide very few abstractions, making the process of building a standardized solution even more challenging.
- **Platform constraints.** Constrained devices are very limited in terms of CPU, memory, available energy and network bandwidth. Especially when considering Class 1 devices (the focus of this thesis), the small available memory requires the use of specific network protocols, libraries, and approaches to reduce the code size. This becomes even more challenging when trying to integrate security and cryptographic libraries. Moreover, these devices are often battery powered and deployed in hostile environments, so that the energy consumption of each component of the architecture needs to be optimized to increase the

lifetime of the deployed solution. The update process must be designed with all these factors in mind, having a small memory footprint, reducing the energy consumption, and being able to deal with unreliable networks, i.e., managing temporary network failures and recovering from the previously reached point.

1.3 Contributions

Facing the identified requirements of security, portability, and platform constraints, this thesis contributes to mitigate the problem of software updates for IoT devices providing the architecture and the implementation of *libpull*, a library suitable to create update systems for constrained devices. It targets Class 1 constrained devices, characterized by approximately 100 kB of ROM and 10 kB of RAM, as defined in RFC 7228 [6]. The solution has been developed in form of a modular library that exposes all the functions required to securely perform the update process, with enough abstraction layers enabling an easy porting to other devices. The design and implementation have been guided by the security, portability, and platform requirements described in detail in an introductory analysis of the update process. The solution contributes to the problem in the following way:

- **Security.** The architecture of the update systems is based on two servers with different goals and functions. The first is the vendor server, in charge of building the update and assert its integrity with a first ECDSA digital signature inserted in a manifest, a set of data used to describe the update. The second is the provisioning server, in charge of effectively distributing the update to the devices and performing a second digital signature to make each update unique to the target device. The transmission of the update is encrypted using DTLS to grant confidentiality and authentication using a server and client certificate. The encryption is performed end-to-end to increase the security, avoiding the use of more powerful gateways to manage security features as often done in architecture containing constrained devices. The signature is verified on the device using a Hardware Security Module (HSM) to safely store the keys and protect them from software attacks.

- **Portability.** The solution has been designed to be agnostic to the OS, to the network protocol, to the cryptographic library, and to the manifest encoding. This has been performed defining the library on top of a set of interfaces, some of them implemented directly into the library (such as the one used to interact with the cryptographic libraries), some of them requiring an implementation by the developer using the library (such as for the network interface).

- **Platform constraints.** The solution has been designed to have a small memory footprint and to be suitable for Class 1 devices even when including the dependencies, such as the network application layer (CoAP) and the cryptographic library. The solution is able to work with different memory types indistinctly, using an abstraction layer that allows interacting with internal and external Flash memory using the same API. Moreover, the library architecture does not limit the possible updates type supported, defined by the characteristics of the OS and of the hardware platform, such as static and dynamic software update loading.

1.4 Limitations

The work in this thesis considers the following set of assumptions:

- The vendor private-key, used to sign the firmware, is not compromised since the server is an offline and protected machine. However, the system uses a two server architecture and a double signature, one performed by the vendor and always trusted, and another performed by the provisioning server, an online and connected machine used to transmit the update to the device. In case the key on the provisioning server is compromised, this will not affect the integrity of the system;

- No physical attacks on the device are assumed. In fact, an attacker able to physically access the memory could modify the code or the encryption keys used to validate the update and thus make the proposed solution invalid. Even when using a Hardware Security Module, a motivated attacker could tamper the device and alter the signature validation process.

With respect to the portability requirement, the implementation of the update system has be only evaluated on two software platforms: Contiki and Linux, and on a single board, the Texas Instruments SensorTag CC2650.

The thesis focuses only on the client part, relying on other solutions for a reliable server implementation. To evaluate the solution, a simple server has been built, but its design and implementation is not considered in the scope of this research.

1.5 Thesis Structure

This thesis is organized organized as follows. Chapter 2 describes the background technologies used in this thesis, useful to understand the following sections. Chapter 3 reviews the related work. Chapter 4 analyzes the update system lifecycle, indicating the role of security in the different phases. Chapter 5 explains the design choices guiding the implementation of libpull. Chapter 6 evaluates the proposed solution in terms of memory footprint, execution time and energy consumption. Chapter 7 concludes this thesis with a summary of its contributions and an outlook on the future work.

Chapter 2

Background

The next subsections describe the technologies used in this thesis to design, implement and evaluate the final solution. Considering the heterogeneity of IoT, we start defining the class of devices focus on this study (Subsection 2.1.1). We explain then the most significant characteristics of IoT and the operating systems used for constrained devices, introducing Contiki, the OS chosen for this implementation (Subsection 2.1.2). We analyze the network protocol stack used in this implementation (Subsection 2.1.3) and, finally, we introduce some cryptographic principles necessary to understand the design choices (Section 2.2).

2.1 The IoT Domain

The IoT is a complex ecosystem of devices and technologies interacting together. As described by H.Suo [7], we can identify four architectural layers:

- *Application layer*: provides the service for the user through API or interfaces;
- *Support layer*: composed of cloud computing and in charge of managing the data needed or collected by the device;
- *Network layer*: responsible for reliable and secure transmission;
- *Perceptual layer*: composed of the device acting as sensor or actuator.

Each layer plays an essential role in the functionality of the final solution. Maintenance and support should be provided at each layer of the architecture and during the whole lifetime of the IoT application. Consequently, security considerations are critical at each layer of the architecture, as a failure in one layer could compromise the other layers and the security of the whole system [5].

This thesis will focus on the security of the perceptual layer, providing a solution to update the software running on the device. However, the update process needs to interact with the other layers as well, and thus requires considerations on the security and reliability of all of them.

2.1.1 Device Classes

The perceptual layer, composed of the device acting as sensor or actuator, is populated by a large set of devices. These can go from small sensors with few kB of memory to complex devices, such as smartphones, composed of many components. The heterogeneity of these devices requires a further classification to better understand the focus of this research.

Covington and Carskadden did a precise classification of IoT devices focusing on security [8]: they classified them in three tiers according to their attack surface, where each tier inherits the characteristics and security issues of the lower tiers. The classification is so composed:

- Tier 1: Entity that contains data and can be uniquely identified, e.g., bar-code and RFID systems;
- Tier 2: Entity that can interact with other objects and the environment, e.g., sensors and actuators;
- Tier 3: Entity that can interact with users, e.g., desktop, laptop, smartphone.

The system complexity and the attack surface grows towards the higher tiers, while the number of devices increases towards the lower ones. Consequently, an attack vector on systems of Tier 2 and Tier 1 is higher compared to an attack vector of Tier 3 [8].

Tier 2 devices are the focus of this research. This Tier is composed mainly of sensors, actuators, and devices with lower resources, lower power capabilities, and reduced cost. The decision of focusing on Tier 2 devices is supported by the fact that these devices reach the best tradeoff between attack surface and number of devices, making them an appealing target for attackers. This choice affects security as well, in both the employed encryption methods and the software updates availability.

Even on Tier 2 devices, we could identify a large set of devices with different characteristics. We can further narrow down the classification, as defined by RFC 7228 [6]:

- *Class 0* devices are very constrained devices that typically rely on a gateway for basic communication;
- *Class 1* devices are quite limited in code space and processing capabilities, but are able to communicate with other devices using specific IoT protocols and without the need of a gateway;

- *Class 2* devices are less constrained in both data size and code size and are able to communicate with other devices using regular IPv4 and IPv6 protocols, similar to the standard network devices.

Table 2.1 indicates the approximative data and code size for each device class.

Class	Data Size	Code Size
Class 0	< 10 KiB	< 100 KiB
Class 1	~10 KiB	~100 KiB
Class 2	~50 KiB	~250 KiB

Table 2.1: Memory size of constrained devices.

This thesis focuses on Class 1 devices, with ~ 10 KiB of RAM and ~ 100 KiB of ROM. Devices that are part of this class are frequently used to build Wireless Sensors Network (WSN), but can also be used to create standalone solutions that need small computational capabilities and low costs, such as wearable devices.

An example of Class 1 device is the one used to test the final solution: the *TI SimpleLink SensorTag (CC2650 SensorTag)* [9] manufactured by Texas Instruments (TI). This device is characterized by ~20 kB of RAM and ~128 kB of ROM. It has capabilities to use BLE or ZigBee/6LoWPAN protocols and is powered by an ARM Cortex M3 processor. The board has two I2C buses, from which it is possible to interact with the ten sensors included, such as humidity, pressure, accelerometer, gyroscope, and magnetometer. The board has also an extension port to support the connection of other modules called DevPacks, that has been used to connect the Hardware Security Modules (HSM) using the I2C bus.

The device will communicate using the IEEE 802.15.4 standard. To communicate with the provisioning server in charge of sending the updates, it will need to communicate with the IP network. This is accomplished using a border router device connected to the host PC through a Serial Line Interface Protocol (SLIP).

The board must be programmed using the *SimpleLink Sensortag Debugger DevPack*, a small board that can be connected on top of the SensorTag using JTAG and that enables flashing and debugging via USB.

From a software point of view, Class 1 devices run small and constrained operating systems, as described in the next section.

2.1.2 Operating Systems for the IoT

There are many operating systems (OS) suitable for Class 1 devices, such as TinyOS, Contiki, RIOT, and FreeRTOS. To target Class 1 devices, an OS should consider constraints in memory, processing power, energy efficiency, and communication bandwidth. One of the most common OS used to build IoT solutions is Linux, however, its requirements are too high for constrained devices since the required Flash and RAM is in the order of MB.

A good comparison of operating systems for constrained devices has been done by Milinkovi et al. [10], who analyzed the memory requirements and the architecture of different OS. One of the first distinctions between the various OS is the kernel architecture, which can be monolithic (as in Tiny OS), layered (as in Contiki) or microkernel (as in RTOS). Another aspect is the programming model and programming language support since some OS do not support standard C, thus limiting the number of dependencies that can be used. Moreover, another aspect is the scheduling strategy, which affects the possibility of operating in real time and have processes with different priorities. The operating system chosen for the evaluation of libpull is Contiki, since it natively supports the chosen board and since applications can be written using the C language.

Contiki is an open source operating system. It has been developed by Adam Dunkels at the Swedish Institute of Computer Science and includes contributions from a large community of developers. It is designed to be lightweight, highly portable, and multitasking. It runs on a multitude of devices and has a low memory usage; a typical Contiki application requires memory in the order of tens of kB of RAM and ROM.

As already mentioned, the Contiki OS is developed in C, and also the applications running on it must be written in plain C. It supports memory allocation using the standard malloc/free, but considering that the code should run on a microcontroller, the usage of static memory is preferred.

The kernel supports multitasking using protothreads, a memory-efficient multithreading programming abstraction. Each time the kernel receives a new event (such as a sensor event), it invokes a protothread to manage it. The protothread is just an abstraction of multithreading since the various threads are not executed in real time. In fact, each protothread yields its execution to return the control to the event loop running in the kernel, which will then send the event to another protothread.

One of the consequences of this execution model is that protothreads are stackless, so they save their state in the private memory of the process. This implies that every variable created inside a protothread will not maintain its content when the control is returned to the main event loop. This approach is completely different from the standard programming approach, where all the variables created in a function remain on the stack until they go out of scope and the function returns.

Contiki allows networking on very constrained devices and supports many network protocols. It supports the uIP TCP/IP stack, which provides IPv4 networking and also uIPv6, which provides IPv6 networking. It supports IEEE 802.15.4 wireless communication with Time-Slotted Channel Hopping (TSCH) and contains an implementation of the Routing Protocol for Low Power and Lossy Networks (RPL).

Many versions of Contiki have been released during the years. Our implementation is based not directly on Contiki-OS, but on Contiki-NG, a recent fork of Contiki with some cleanup and improvements. Contiki-NG has been used to build the update agent and also to build the bootloader. The goal of building the bootloader on top of Contiki-NG is to allow a smooth integration with all the other platforms supported by Contiki-NG itself.

2.1.3 Protocol Stack

Creating a technology that facilitates the interconnection of IoT constrained devices has been challenging. The platform constraints required the effort of many working groups to define protocols and standards that enable the reliable interconnection of millions of objects. The need for a new set of protocols was clear, as are needed to deal with the following requirements:

- **Low energy consumption.** IoT devices are typically battery powered and thus require a low-power communication stack. In fact, the extensive number and the inaccessibility of IoT objects make charging or replacing batteries an expensive and not always feasible operation;
- **Critical deployments.** Critical deployment environments require a high reliability of the communication. A communication failure in a healthcare application could potentially harm the life of people relying on this technology;
- **Interoperability.** IoT devices will need to interact and communicate with millions of already deployed devices and networks. Having an interoperable stack could reduce the needs of protocol translations and increase the adoption of this new technology.

Many standards such Zigbee or BLE satisfy the previous requirements. Another common protocol stack for IoT is composed by *IEEE 802.15.4* for the Physical (PHY) and Medium Access Control (MAC) layers, *6LoWPAN* as IPv6 adaptation layer, *RPL* as routing protocol, *DTLS* to integrate security and *CoAP* as the application layer.

In the next sections, we will briefly discuss this set of protocols, as they are the base for many constrained devices and will also be used to test the final solution.

- **IEEE 802.15.4**: is a technical standard which defines the operation of low-rate wireless personal area networks. The power-consumption of IoT device is massively affected by the radio components, as they usually are the more power hungry. The radio consumes a considerable amount of energy when enabled. It does not consume energy when it is disabled. The challenge of a good communication stack is to reduce the time in which the radio is enabled. The radio duty cycle is the percentage of time the radio is on, and it is a good indicator of the power consumption of the device. An energy-efficient communication stack has a duty cycle lower than 1% [11]. The IEEE 802.15.4 standard defines 16 frequency channels, and the radio can arbitrarily send and receive on any of those channels. The first standard defined the communication as fixed on a single channel. It became soon visible how this caused reliability problems due to RF interferences. A better approach was time synchronization and channel hopping, allowing a low duty cycle and high reliability. In 2010 a new extension of the standard (IEEE 802.15.4e), introduced the Time Synchronized Channel Hopping (TSCH), that uses a frequency diversity that mitigates the effect of the interference with other RF networks. The use of TSCH also increases the network capacity, because nodes can transmit at the same time using different channels.

- **6LoWPAN**: IPv6 over Low-Power Wireless Personal Area Networks is a standard designed by the IETF 6LoWPAN working group in RFC 4944 [12]. The idea behind this protocol is to apply the Internet protocols used on conventional devices also on small and constrained devices. This approach grants scalability, global reachability, and the use of a well-tested technology such as the IP network as a background. To reach this goal, the IPv6 protocol was compressed to fit the required size. IPv6 was the more indicated protocol, allowing interconnection of millions of devices. Thus, the 6LoWPAN WG defined encapsulation and header compression mechanism to allow the transmission of IPv6 packets over IEEE 802.15.4 frames. The general approach used is to reduce the size suppressing redundant information in the IPv6 packet header that can be derived from other layers of the communication stack.

- **RPL**: The Routing Protocol for Low power and Lossy Networks is the routing protocol used on top of 6LoWPAN networks, and is standardized in RFC 6550 [13]. It can quickly build network routes, distribute routing information and efficiently adapt the topology. When forming the topology, each node sends a DAG Information Object (DIO) to all its children. When a child decided to join the DAG, it should pass the DIO further to its children. The DIO contains a rank that is increased when the child joins the DAG, preventing routing loops and helping the nodes to distinguish between parent and siblings [11].

- **DTLS**: Datagram Transport Layer Security is a communication protocol used to provide security to datagram based applications with a lightweight approach and considering the requirements of constrained devices. Based on the TLS standard, DTLS is also designed to allow confidentiality, integrity, and authentication to the communication. Differently, from TLS, DTLS was designed for datagram protocols and cannot rely on the functions of the underlying protocols to deal with packet reordering and packet loss. The handshake protocol represents the first phase of setting a DTLS connection, and is also the most expensive phase in terms of computation and memory resources. It starts with a Hello message and continues with the negotiation of the employed algorithm using the concept of the cipher suite, a string indicating the various algorithms used to preserve integrity, authenticity, and encryption. When client and server agreed on a cipher suite, they can proceed to exchange a shared secret. The algorithm used is also indicated by the cipher suite. One of the most used ones is the Diffie-Hellman algorithm. The handshake protocol is implemented very similarly to TLS, but it also integrates a cookie exchange to prevent denial of services attacks, a timer to handle message loss, and some modification to the header to avoid IP fragmentation (since it is already performed at the DTLS level). The use of the cookie is necessary as datagram protocols are very susceptible to Denial of Service attacks. In fact, considering that no TCP-like connection is required, an attacker could initiate a series of handshakes with a low effort, forcing the server to allocate state for the each request.

- **CoAP**: the Constrained Application Protocol is a specialized protocol for constrained devices standardized by IETF RFC 7959 [14]. It is designed to enable constrained devices to communicate using the same REST approach used on the Internet, thus inspired from the HTTP protocol. It has been designed to enable communication between nodes of the same Personal Area Network (PAN), but also with devices on the standard Internet. In fact, the CoAP protocol can be easily translated to HTTP using a specific proxy, to allow integration of constrained devices with the Web. CoAP implements a subset of the functions of HTTP and uses the same REST approach based on the GET, POST, PUT, DELETE verbs. Moreover, CoAP also defines the possibility to subscribe to a topic, useful for IoT devices to reduce polling and unnecessary communication. To reduce the size, CoAP uses binary encoding and is based on two message types: request and response. The CoAP standard defines either UDP or DTLS as the underlying protocols. The protocol has been designed in a way that a message should fit in a single packet minimizing fragmentation. In case the transfer of a higher number of bytes is necessary, the Block-Wise option can be used. This allows transferring a higher number of bytes by dividing them into blocks.

2.2 Security Background

In this section, we introduce some cryptographic fundamentals necessary to understand the design choices of the final solution. We also introduce the reader to the security properties that will be required by libpull and give an introduction to the Hardware Security Modules in general and the specific device used in our implementation.

2.2.1 Cryptographic Fundamentals

The method to securing data and make sure that only those who are allowed can access them is called cryptography. It poses its foundations on advanced mathematical principles and, according to the security property we want to ensure, it can be implemented with different methodologies. For example, to grant confidentiality, encryption can be used. To grant authentication, digital signature or Message Authentication Code (MAC) can be used. The use of cryptography helps mitigating many security threats and plays a critical role in the security of an update system.

Encryption

The first important cryptographic primitive is encryption. The goal of encryption is to protect data from unintended parties. It is a process composed of two steps: encryption, and decryption. The encryption phase converts some data, called plain text, to another form, called encrypted text, using an encryption algorithm. In this way, the encrypted text can be sent through an untrusted channel and its content protected from an eavesdropper. The other end receiving the data needs to decrypt it to obtain the original plain text. The conversion between a format and another is based on a secret, called the secret key. There are two main approaches: symmetric key encryption and asymmetric key encryption. These two methods are very different in both theory and implementation.

Symmetric encryption is the most straightforward kind of encryption. It is an old technique and bases its security on the secrecy of a shared secret between the parties. Many symmetric algorithms have been developed, such as RC4, DES, and AES but the most widely used today are AES-128, AES-192, and AES-256. In symmetric encryption, both parties share the same secret, called key. The storing and distribution of this key is typically the biggest problem of symmetric encryption; thus it is used in combination with asymmetric encryption.

Asymmetric encryption, also known as public key cryptography, is a relatively new method compared to symmetric encryption. It is not based on a shared secret between the parties but uses instead two keys, a private and a public one, basing its security on the secrecy of the first one. The private key is used by the sender to encrypt the data, while the receiver uses the public key to decrypt the data. The key concept is that the only key that must be kept secret is the private one, allowing to share the public key also using untrusted channels. Many algorithms have been developed, such as:

- RSA (stands for Rivest, Shamir, and Adleman) is one of the first asymmetric cryptography algorithms. It is widely used today for secure Web communications and to integrate security into many other protocols, including IMAP, and FTP. RSA poses its security on the robustness of the integer factorization problem, which states that is very difficult to factorize large numbers [15].

- ECC (Elliptic Curve Cryptography) is an asymmetric cryptography algorithm based on the algebraic structure of elliptic curves over finite fields. It is considered secure on the assumption that it is infeasible to find the discrete logarithm of a random elliptic curve element with the knowledge of base point represented by the public key. ECC requires a smaller size compared to RSA which results in faster computation, lower power, memory, and bandwidth usage, making it a suitable protocol for constrained devices.

Asymmetric encryption requires more computational power and more time compared to symmetric encryption. To take benefit of both algorithms, a hybrid approach is typically used, based on asymmetric cryptography to exchange a secret and on symmetric cryptography to continue encrypting the data using that secret.

Key Exchange

The goal of a key exchange algorithm is to allow two users, each holding a public-private key pair, to agree on a shared secret over an untrusted channel. The most used key exchange algorithm is Diffie–Hellman (DH). In case elliptic curve cryptography is used to perform the key exchange, we talk about Elliptic Curve Diffie–Hellman (ECDH). The shared agreement between the nodes can be then used directly as a key or to derive another key.

Cryptographic Hash Functions

Another important primitive used in cryptography are hash functions. These functions take as input an arbitrary size of bytes and produce as output a small and fixed number of bytes denominated the hash. The operation is irreversible, meaning that, starting from a hash, is not possible to get any information on the original data. Moreover, hash functions are designed to give a completely different hash starting from different data. This property is called collision resistance, and in the past, many algorithms were deprecated due to the possibility of creating collisions, such as *md5* or *SHA-1*. The most widely used family of hash functions are the Secure Hash Algorithm 2 (SHA-2). They can work with a hash size of 224, 256, 384 and 512 bits.

Hash functions are essential for many other cryptographic primitives, such as digital signature. In fact, some algorithms are able to work only on a finite set of bytes and requires to preprocess the data with a hash function to accomplish their goal.

Digital Signature

The digital signature is a process that provides integrity and authentication of the signed data. Like a real signature, digital signatures are unique to the signer and can be used to verify the integrity and the authenticity of the data.

The digital signature process can be divided into two phases. In the first phase, the message is signed using a signing algorithm and a key. In the second phase, the message is verified, using a process called signature verification. If the verification is successful, one can assert that the data has not been modified and can even state the authenticity of the signer.

Using an asymmetric algorithm to perform digital signature is beneficial, as it does not require protecting a shared key on the client, allowing in the case of the software updates an easy storing of the public key on the device memory. The asymmetric algorithms that can be used are RSA, DSA (digital signature algorithm) or ECDSA (Elliptic Curve DSA). The last one is the one chosen for our implementation, due to some smaller size of the keys and signature.

Different algorithms require a different key size to provide the same level of security. This is presented in Table 2.2:

Symmetric	ECC	DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Table 2.2: Same level security of encryption keys.

True Number Generator

Another essential concept in cryptography are random numbers. The unpredictability of the numbers enforces the strength of algorithms that need a nonce in input, an arbitrary number that must be used only once. Generating random number is not easy, since it requires each number not to be related to the number generated before or any other property available to the attacker. For this reason, standard computers generates random number collecting randomness from multiple sources, such as timely of key presses or network traffic, and gives them as input of a Pseudo-Random Number Generators (PRNG). However, this is not always possible on embedded devices where the entropy sources are lower. For this reason, a specific hardware modules in often included in the device, which is called True-Random Number Generator (TRNG).

2.2.2 Security Properties

With a basic knowledge of the cryptographic primitives it is possible to understand the security pillars of information security. The understanding of these concepts is essential, as they will be widely used in the next chapters of this thesis.

Confidentiality

Confidentiality ensures that, except for the authorized entities involved, the data exchanged during communication is kept confidential. This is crucial for IoT applications, as any failure would seriously threaten user's privacy. To grant confidentiality, cryptographic algorithms are generally used to cipher data. Doing so, even if the exchanged data eavesdrops, the attacker will not be able to access its content.

In contrast to the security by obscurity principle, Kerckhoffs principle states that a cryptosystem should rely on the secrecy of the keys. In fact, this principle assumes that an attacker is able to access and master the cryptographic protocol. In the IoT world, the confidentiality property is often not granted, as many vendors use to deploy devices with the same shared key that, once discovered on a device, makes all the encryption made by devices with the same key useless.

Integrity

Integrity ensures that information is not modified, accidentally or purposefully, without being detected. This is valid for the data sent through the network but also for the firmware and the programs loaded by the device. Data should not be modified in an unauthorized and undetected way. There are various ways to verify the data integrity. Often a hash function is used, to create a digest for a particular block of data. The hash value is calculated by the sender and the receiver. If both results match, then the integrity property is granted. Considering that the data and the hash functions are usually sent through the same untrusted channel, it is necessary to grant that the digest has not been altered using techniques such as asymmetric encryption. A simpler method to verify the integrity of data is using a cyclic redundancy check (CRC), that could be used to detect if the data accidental changed. However, this is not suitable in case we want to protect the data from malicious changes, where also well-known hash functions such MD5 or SHA1 are now considered deprecated. The hash functions that should be used are the ones from the SHA-2 family, such as SHA-256, or SHA-512.

Authentication

The authentication property ensures that the source of data has a known identity or endpoint. The process of authentication is normally performed showing to the device requesting authentication a proof of identity, such as a shared and secret key, a user-id and password, or some biometric data. This property is normally implemented with a digital signature or with authenticated encryption. In a client-server communication, the server is normally the only end begin authenticated. However, in some specific applications, such as software updates, it becomes important to authenticate also the client. Authentication is also used to prevent Man In The Middle (MITM) attacks, where a malicious user eavesdrops the connection and acts as one of the involved parts. If a device is not able to correctly authenticate the other, for example by recognizing an invalid certificate, it will believe that the connection is secure, even in presence of a MITM attack.

Availability

The availability property ensures that data is available when needed by authorized entities. This implies that the communication system has to remain functional despite security attacks (i.e., DoS attacks) and hardware failures. A way to increase the availability and also the reliability of IoT systems is to provide redundancy for critical devices and services.

Non repudiation

The non repudiation property ensures that an individual or system cannot later deny having performed an action. This property is important in many IoT applications, because it supports the transfer of responsibility, liability, and culpability needed to allows some devices to perform the autonomous operation and independent decision making. The decisions of smart objectscan, in unfortunate circumstances, cause harm to humans or things. In this case, the non-repudiation property ensures that we are able to attribute responsibility for certain events to non-human agents and it also needs the identification of the device to attribute the non-repudiable action to the right entity. In our particular application, this property is not necessary.

2.2.3 Hardware Security Modules

A Hardware Security Module (HSM) is a dedicated cryptoprocessor able to perform cryptographic operations and securely store keys and other critical security parameters (CSPs). HSMs are typically used on servers to reduce the computation power required by the main CPU, especially when using long asymmetric keys. These modules are sold in the form of smart cards or external device that can be attached to a server [16]. In our case, we consider solutions explicitly designed for IoT products, available on the market as small integrated circuits that can be integrated into the final solution.

The Kerckhoffs's principle states that the security of a cryptographic algorithm must be based only on the secrecy of the key. HSM increases the security of a generic software cryptographic solutions being able to safely store the keys, protecting them from software attacks and, in some cases, also from hardware ones. Keys can be generated inside the HSM, reducing the possibility of being compromised, or can be generated on a host machine and successfully loaded into the HSM during the personalization phase. When the key is stored inside the device, it is never exposed to the main MCU and can be used internally to perform cryptographic operations. The security of the encryption key is very important. Indeed, poor key management leads to the disclosure and compromise of cryptographic keys, making all the effort of integrating security countermeasures ineffective.

Hardware Security Modules also provides other features, such as True Random Number Generator (TRNG), solving a critical problem in the embedded device with few entropy sources. Some HSM facilitates also the process of authentication by calculating a Message Authentication Code (MAC) using a received challenge and a secret key stored in an internal memory slot of the HSM.

There are many HSM available on the market. For this thesis, we use the Atmel CryptoAuthentication devices [17]. These devices implement cryptographic functions such as AES, SHA256, HMAC, and ECC. After testing the possibilities offered by the devices, we found in the ATECC508A the more useful for our architecture.

The ATECC508A offers ECDSA and ECDH based on the NIST-P256 elliptic curve. It also includes a TRNG and secure key storage for up to 16 keys. It is connected to the main MCU via I2C and supports connection up to 1MHz. The usage of the I2C bus is limited to just a small amount of data to perform ECDSA operations. The connection between the MCU and the ATECC508A is a weak link, as the data exchanged in plain text through this channel could be easily faked with a physical attack. To solve this problem, the ATECC508A allows receiving encrypted commands and includes an HMAC for the response, which can be used by the main MCU to validate the returned value.

Chapter 3

Related Work

The need for software updates is not something new to the IoT world. It has been already encountered in complex systems like servers, computer, cars, or small but sophisticated devices like smartphones. The most similar device type to our target Class 1 devices are Wireless Sensor Networks (WSN), however, all the software updates research and project for these devices rarely targeted security explicitly, focusing mainly on energy efficiency and network bandwidth. Each of the previous device classes has different goals and requirements from the update system and each solution has its advantages and disadvantages. This chapter investigates the software update process with a top-down approach, moving from the more powerful platforms to the more constrained ones, to finally reach the Class 1 devices target of this thesis. For each class, we will briefly explain how the update is performed to understand the best approaches that can be ported to constrained devices.

Computer Software Updates

Considering servers and computers, software updates are typically performed at a package level, updating the necessary program, their dependencies or the kernel [18]. Many package managers are present today, and each OS and Linux distribution tends to have a different one with slightly different characteristics. Sometimes, the updates are also managed by the application itself that, running a parallel program that periodically checks the presence of an update and updates itself. This approach is normally used by browsers [18]. The update managers for computers do not have to deal with hardware constraints and work normally at a high level using the abstractions provided by the OS. Moreover, all the modern package managers download the update from a trusted endpoint using an encrypted connection and verify the digital signature. This kind of updates and platforms are very different from the devices targeted by this study, where the update system does not download a package, but a whole firmware and does not have any abstraction to access the memory, but needs to store the update using raw memory access.

Smartphone Software Updates

Another device class that, at least in physical size, is more similar to the devices target of our study are smartphones. From the early stages of their development, smartphones included an App Store, a program allowing the user to browse and install new applications, but also managing the update process. The functional and security features added to the App Store and to the operating system update agent increased over time, often being target of attacks used to perform privilege escalation and execute untrusted code on the device. On mobile devices, the update agent is able to work autonomously to update the applications but needs the user authorization to update the whole system [19]. This increases the reliability of the process since the user is aware of the update and can try to recover in case of failures. However, it requires a user interface to interact with the user and this is often not possible on constrained devices, that must work in a completely autonomous way and are often deployed in hostile environments. A lot of research work has been done on update system for mobiles devices but we believe the most advanced and tested solutions are deployed in the two most used operating systems, Android and iOS. One of the challenges that have been solved by the iOS update system is to grant the update freshness, avoiding a downgrade or the possibility to update a device not to the oldest firmware. This is performed using an *SHSH blob* [20], a small piece of data that contains the device identity and makes the update unique for each device request. This approach has been also included in our solution and is explained in Chapter 4.

Embedded Systems Software Updates

Another class of devices more similar to our target are embedded systems. The solutions available target security, small memory footprint and power efficiency, but at a higher level than the one required. Typically embedded devices running a Linux based OS (such as the Yocto project [21]) have an internal memory in the order of tens of MB, where constrained devices are in the order of ˜100 kB. There are several open source solutions developed for embedded systems, such as *Mender* [22] and *SWUpdate* [23].

A lot of research has been done to update the embedded devices used in the automotive field. In this particular application, in fact, security is always put at first since it could affect the safety of drivers. Moreover, considering that the size of software included in today's car is always increasing, the need of sending software updates is much higher and without, an Over The Air (OTA) update system, requires the auto to be recalled to the workshop, with high costs for the device manufacturer [24]. As an example, Fiat-Chrysler in 2015 had to recall 1.4 million Jeep Cherokee to fix a security vulnerability allowing an attacker to remotely manage the car [25]. In 2008, Nilsson et al. introduced a protocol for FOTA in vehicles which ensured data integrity, authentication, and confidentiality. However, their analysis did not address how to safely store the verification keys. Jurkovi et al. [26], discuss the update process for constrained devices without an operating system, focusing on the transmission phase and the memory footprint. However, the solution did not target the security of the update process.

Wireless Sensor Network Software Updates

As already indicated, update systems for WSNs represent the most similar architecture to Class 1 devices, and could be hence potentially suitable. In fact, Class 1 devices are often used to build WSN networks. However, the research on WSN update systems focuses mainly on transmission protocols, traffic reduction, and software optimization, without often considering security as a primary requirement. In this class of applications, the update process is often called *reprogramming*, since it is used to add new functions and reprogram the device behavior rather than fixing security vulnerabilities. A good overview of WSN updates is done in [27], where the author analyzed the generation, propagation, and activation phase of four systems, such as *MOAP* [28], *Maté* [29], *Impala* [30], and *Deluge* [31]. These systems are suitable for Class 1 devices, but are not designed with security in mind. In fact, *MOAP* focuses on the performance of the propagation protocol; *Maté* concentrates on the propagation and mobile-code activation aspects; *Impala* concentrates primarily on the propagation of the update and provides some sanity check of the newly loaded code (only valid memory accesses); whereas *Deluge* concentrates on the efficient propagation via incremental updates. All these solutions tried to optimize the propagation protocol minimizing the traffic. A possible approach is to use delta updates, and can be performed in many ways, such as dividing the firmware into sections and update just some of them [32] or shipping patches containing only the differences between the old and the new firmware version.

A research on WSN updates focusing on security has been performed by Cheng et al.[33]: to avoid the spread of malware inside the WSN the author propose an architecture where the intermediate and more powerful nodes are in charge of detecting malicious code. Once a malware is detected, the patching phase starts by updating first the intermediate nodes to avoid spreading the malware to all the device in the network. This solution is not directly related to the update process, but it is interesting, since it tries to move part of the security solution to the more powerful nodes of the network. This path has been explored, but discarded since, in our architecture, Class 1 devices could work as standalone devices and are not necessarily integrated into a WSN.

Constrained Devices Software Updates

Our target device class requires an update system that includes many of the features discussed previously and inherits many approaches from some of them, such as security from the computer and smartphones software updates, portability from the embedded systems software updates, small memory footprint, and low energy consumption from the software update systems for WSN.

Currently, no generic update system for IoT Class 1 devices has all the previous characteristics, and there is no open source solution that is suitable for Class 1 constrained devices and that can be integrated by vendors when producing devices for the Internet of Things.

Chapter 4

Software Updates Analysis

Software updates for constrained IoT devices are very different from the updates of ordinary computers, and many different implementations are possible according to the platform and application needs. There is more that one correct approach on how to structure the memory and how to transmit the firmware, as the implementation needs to deal with the hardware and software capabilities. A device may have just the internal flash or could also include an external memory. From the software point of view, the operating system could allow dynamic loading or not. For this reason, it was evident that the approach of creating an application was limiting the possibilities of the solution and that a more modular approach was needed. We implemented the solution as a library, libpull, which provides enough abstraction to be extended and migrated to other software and hardware platforms.

The process of software updates has been identified as one of the most critical threats by many software security research work specific on IoT [5]. Considering that the update system can install or override the code running on the device, an attack able to compromise this process would affect the integrity of the running application and allow to inject malicious code, completely changing the behaviour of the device. We saw in the literature many attacks exploiting the software update process that are also able to create a worm and infect all the devices in the neighborhood [34].

In this chapter, we start introducing the problem of software updates defining the terminology and the general components. These are required to understand the next sections, where threads and requirements to secure the update process are analyzed. We analyze then the requirements of the update system for Class 1 devices, and also the portability requirements that our solution should satisfy. Finally, we summarize the designed architecture discussing the life cycle of an update, moving from the vendor server to a correct and secure installation on the device. Considering that not every system will require the same level of security, we provide four security configurations that include in an incremental way the identified security requirements.

4.1 Software Updates Essentials

Providing software updates is not a new challenge and affects every connected system previously developed. Software updates are needed for two main reasons:

- Fix bug or vulnerabilities;
- Integrate, update or remove features.

Depending on the reason for performing the update, we could have different update types. Fixing bugs and vulnerabilities is usually performed with minor updates, based on little changes of the codebase. The integration of new features instead, usually requires substantial changes to the codebase and is done in a major update. Both types of updates could introduce new bugs and vulnerabilities, consequently providing updates is not a single operation but rather a process that must be reiterated over time to increase the quality and security of the solution.

Each update is identified by a version, and by a number that identifies a specific point in the history of that software. In case the version of the downloaded update is higher than the one that must be updated, the process of updating is also called upgrade. Otherwise, in case the new version is lower, then the process is called downgrade.

Updates are performed in different ways according to the platform where they are implemented and can be categorized in terms of granularity. This property indicates the dimension of the software that needs to be updated and its relation to the overall system. Smartphones software updates show a clear evidence of this property. In fact, it is possible to have a fine granularity update, upgrading just a single application, or a coarse granularity update, upgrading the whole operating system. When considering IoT devices, we identified three possible granularities:

- *Package-based updates*: Used to update just some parts of the application. This concept is well known on Linux based embedded systems to update applications and dependencies in a modular way. Package managers are used to regularly check the dependency version, download and apply new updates.
- *Modular updates*: Based on the concept of a modular operating system, where the kernel, the libraries, and the applications are physically stored in different memory segments. In this case, it is possible to update just the changed section, leaving the others unchanged. This approach typically needs to relink the various modules before their execution.
- *Image-based updates*: Used to update the whole image. Embedded systems usually rely on this method since updating the full image provides atomicity to the process, avoiding compatibility issues and ensuring that an update is entirely installed or not at all. The atomicity property also ensures that the application behaves the same way in the test and production environment [4].

Each update granularity has its advantage and disadvantages. However, when considering Class 1 devices, we identified only the modular and image-based update as suitable, and only these two models will be considered during the analysis.

The update can be transferred to the device using different methods. It can be transferred via a physical device, such as removable USB stick, or using Over The Air (OTA), using a wireless network to receive the update. Considering the connected nature of IoT and the requirement to perform the process autonomously, we consider only the OTA updates as a goal for our solution and exclude the physical transfer from the supported possibilities. The OTA update can be transferred using different approaches, which we can classify according to the network topology used:

- *Client-Server approach:* The update is transferred from the server to the client without middle devices involved in the process. The middle devices can be involved in the transfer of the data acting as network nodes, but do not perform any action on the update, having a passive role in the update process;

- *Mesh approach:* The update is transferred from the server to the client or from a middle device to the client. The middle devices can receive the update and resend it to the neighbor nodes, taking an active role in the update process.

We excluded the mesh approach from our analysis, since the possibilities offered by the Client-Server approach to integrate security features are higher and allow the server to manage the process entirely. Moreover, having each node transferring the update to the nearest node makes the process of shipping updates more complicated in case the server wants to send it to a subset of devices for testing or other purposes. We can still break down the Client-Server approach in two classifications, according to the propagation direction of the update from the server to the client.

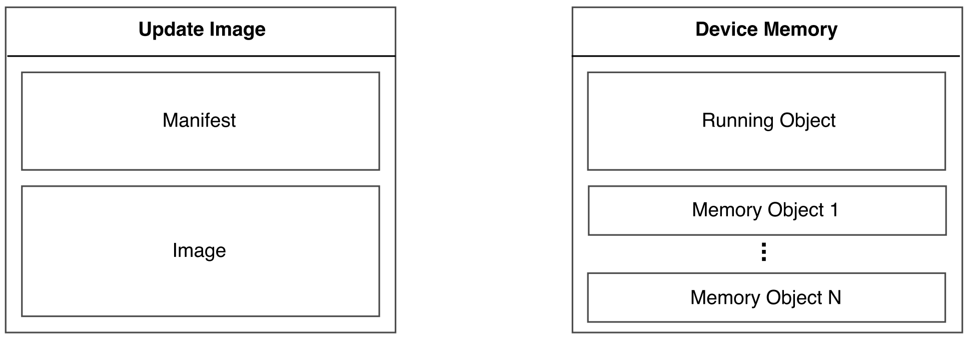
- *Pull mode.* The client starts the process of updates, periodically contacting the server to check the availability of an update. The polling frequency affects the vulnerability window and the energy consumption;

- *Push mode.* The server starts the process notifying to the device the presence of a new firmware. This mode is intrinsically faster than the pull one, since it removes the polling delay and consequently decreases the vulnerability window. However, it needs to find the device in a receiving state and also implies its reachability from the public network, increasing the possibility of being compromised.

We designed our solution focusing on the pull mode, where the client periodically checks the presence of an update contacting the server. This allows to decrease the complexity of the server and makes each client responsible for its own update.

4.1.1 Update Terminology

To understand the following sections is necessary to agree on some terminology and concepts, used to analyze and describe the update process.



(a) Update image. (b) Device Memory.

Figure 4.1: Update terminology and relations.

We first define the terminology to describe an update:

- *image*: the software that will be executed on the device. It can be a firmware, in case it must be loaded by the device MCU, or could be a software module that can be loaded at runtime by the OS;
- *manifest*: the set of data related to the image, describing its size, its version and including all the data used for cryptographic verification;
- *update image*: the union of image and manifest representing the update. This is the data that must be generated and transmitted by the server to the client.

The previous concepts are defined in Figure 4.1a. We also define the terminology to describe how an update is stored in memory as:

- *memory object*: the section of memory used to store the update. This can be a file, a segment of the internal or external flash, a device or any other memory abstraction defined by the user of the library;
- *running object*: the currently running image;
- *device memory*: a generic memory of the device that contains one running image and one or more memory objects.

The previous concepts are illustrated in Figure 4.1b.

4.1.2 Software Updates Components

To solve the identified problem of many vendors not having the resources to manage the update process, we considered two servers, possibly managed by different entities. Considering that the update process should be performed completely automatic to increase the security reducing the vulnerability window, we do not consider the user as an active component of the process.

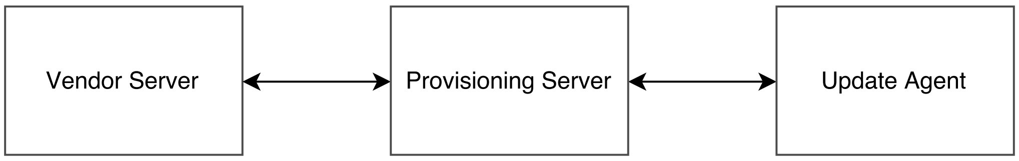


Figure 4.2: Update process components.

We identified three distinct components:

- The **vendor server** is the server owned by the device vendor. This server is the first point where the update is built and thus, it is used to assert its integrity and authenticity. This server perform the following actions:
  - Builds the image;
  - Generates a manifest file;
  - Generates the update image;
  - Sends the update image to the provisioning server.
- The **provisioning server** is the server in charge of communicating with the device. It may be managed by the device vendor or not. It performs the following actions:
  - Notifies updates availability;
  - Updates the manifest;
  - Sends the update image to the device;
  - Logs the device status.
- The **update agent** is the code running on the device with the goal of getting the newest firmware available. It performs the following actions:
  - Checks the presence of updates;
  - Receives the updates;
  - Validates the update;

– Activates the update.

The update image moves from the vendor server to the provisioning server and finally to the device, where – if the validation passes – it becomes the new running object.

4.2 Software Updates Security

In the previous sections, we decomposed the update application identifying the involved entities and interactions between them. They can be considered as the entry point of our system and guide us to identify the associated threats. In this section, we will perform a threat analysis and then sort the identified threats according to their risk. Based on this, we will define the requirements of the software update process and four possible configurations, explaining the risks associated with each of them that may be acceptable in some applications.

4.2.1 Threat Model

The STRIDE model is a threat classification model that helps to identify and classify the threats of a systems in six categories [35]. Each category can be mapped with a security property identified in Section 2.2. I will briefly describe the six categories, and indicate the associated property in Table 4.1:

- *Spoofing.* Attack aiming to illegally impersonate another entity with the goal to gain access to resources it should not have access;
- *Tampering.* Attack aiming to maliciously modify data, such as authentication keys, or even running code;
- *Repudiation.* Attack aiming to perform illegal operations without the possibility of being tracked;
- *Information Disclosure.* Attack aiming to exfiltrate information;
- *Denial of Service.* Attack aiming to deny access to valid users or make the resource unusable;
- *Elevation of Privilege.* Attack aiming to gain privileged access to a resource;

We will analyze now the identified threats. They will be classified according to the previous categories, and for each of them, a mitigation technique is provided.

Threat 1: Device key access. Classification: Elevation of Privilege.

The attacker can obtain the key stored into the device. With it, the attack can generate a valid signature and install malicious code on the device. This attack does not affect the other components.

Mitigation: use public key cryptography and store the key in read only memory or use symmetric key cryptography storing the key using specific hardware modules.

Threat 2: Device key modification Classification: Tampering.

The attacker can exploit a software attack on the device application or the update agent to modify the device key. This allows the attacker to generate a valid signature matching the malicious inserted key.

Mitigation: store the key in a read-only region of the memory or protect it using specific hardware solutions.

Threat 3: Vendor server compromise. Classification: Elevation of Privilege.

The attacker can remotely access the vendor server. The attacker can inject malicious code into the image or obtain the signing key, allowing to generate an update image containing malicious code. This attack affects the vendor server and the device as well.

Mitigation: The vendor server should be a protected and off-line machine, used only to generate the manifest and the update image. The transmission of the update image to the provisioning server should happen using an out-of-band channel.

Threat 4: Provisioning server compromise. Classification: DoS.

The attacker can remotely access the provisioning server, from which he can send invalid images or notify the update agent of a new update. This attack involves the provisioning server and the device as well.

Mitigation: The update agent should be able to detect a failure during the update and include a timer able to delay continuous invalid updates.

Threat Category	Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non repudiation
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorization

Table 4.1: Relation between threat category and security property.

Threat 5: Old vulnerable Image. Classification: Elevation of Privilege

An attacker has access to a valid but vulnerable update image. In case a device has a running image version lower than the vulnerable update image, the attacker can successfully install it on the device since it will pass the validation process. The impact of this attack depends on the vulnerability exploitable in the old image.

Mitigation: the provisioning server must sign the image including a nonce generated by the device and its device ID.

Threat 6: Invalid platform or application. Classification: DoS.

The attacker sends an update image with a valid signature but for another platform or application type. This will lead to the device installing the invalid update image, making the device unavailable after the loading phase.

Mitigation: include information on the platform and the application type in the manifest. The device must validate their data by comparing it with information stored in read-only memory.

Threat 7: Deletion of running image. Classification: DoS.

The attacker can force the download of the update in the memory object containing the running image. At some point of the update image transmission, this will lead to the brick of the device.

Mitigation: The update agent must not receive any instructions on the storage location from the network, but it must decide autonomously the final location of the update by analyzing the content of the memory objects.

Threat 8: Device cloning. Classification: Information Disclosure.

The attacker can create a copy of the device software and receive the updates from the server.

Mitigation: Authenticate the client using a certificate. Store the private key in a hardware security module.

Threat 9: Invalid size. Classification: DoS.

The attacker can send a wrong size of the update image. In case the size is bigger than the correct one, the update agent will overwrite another memory object and potentially invalidate the running image. In case of smaller size, the update image will be corrupted.

Mitigation: implement protection algorithms to avoid overwriting another memory object.

Threat 10: Higher version. Classification: DoS.
The attacker, spoofing the network, can send a version that is higher than the running image version. This forces the device to download the running image, but

the invalid signature will prevent it from loading it. This attack performed multiple times could lead to a Denial of Sleep attack.

Mitigation: authenticate and encrypt the connection between the provisioning server and the device using a strong encryption algorithm.

Threat 11: Provisioning Server redirection. Classification: DoS.

The attacker can redirect the request to an invalid server, blocking the subscription request performed by the device.

Mitigation: Authenticate the connection between the provisioning server and the client to recognize a MITM attack.

Summary

The previously described threats are summarized in Table 4.2. Threats have been sorted from the highest to the lowest risk with the goal to prioritize the mitigation effort.

Code	Description	Classification
T1	Device key access	Elevation of Privilege
T2	Device key modification	Tampering
T3	Vendor server compromise	Elevation of Privilege
T4	Provisioning server compromise	Denial of Service
T5	Old vulnerable Image	Elevation of Privilege
T6	Invalid platform or application	Denial of Service
T7	Device cloning	Information Disclosure
T8	Deletion of running image	Denial of Service
T9	Invalid size	Denial of Service
T10	Higher version	Denial of Service
T11	Provisioning Server redirection	Denial of Service

Table 4.2: Identified security threats.

The risk of each threat has been calculated using the formula:

$$\textbf{Risk} = \textbf{Likelihood}\ x\ \textbf{Impact}$$

where the *Likelihood* is the probability of this type of attack to happen and the *Impact* is the potential damage that it can produce. To make the analysis less subjective, qualitative values have been used, such as “High”, “Medium”, and “Low”.

4.2.2 Security Requirements

The threat model analysis illustrates the identified threats of this process. A mitigation technique was presented for every threat and we will now formalize the various mitigation techniques into security requirements that will represent the basis for the design and the implementation of the final solution.

Security Requirement 1: Double server architecture.

Two physical servers must be used for the application. The vendor server must be protected and we assume it is an off-line server used just to generate the manifest and sign the image. This strong assumption makes sure that the vendor is not compromised. The provisioning server, instead, must be an online server in charge of communicating the update agent in a secure way.

Security Requirement 2: Public-Key based digital signature.

Unless good protection measures are used, symmetric key cryptography represent a high threat for IoT security, considering that it requires a shared secret between the client and the server and that the client can be possibly physically accessed by an attacker. A public-key based digital signature does not require any shared secret and the private key can be safely stored in the vendor server. Moreover, this reduces also the costs of key distribution, since every device can use the same key and does not need key diversification like with symmetric cryptography.

Security Requirement 3: Double server signature.

The signature must be performed in both the vendor and provisioning server. This allows to make sure that an attacker able to compromise the provisioning server will not be able to run code on the device, since the verification requires also the check of the vendor signature. Moreover, it allows the provisioning server to

Code	Description	Likelihood	Impact
T1	Device key access	High	High
T2	Device key modification	High	High
T3	Vendor server compromise	Medium	High
T4	Provisioning server compromise	High	Medium
T5	Old vulnerable Image	Medium	High
T6	Invalid platform or application	Medium	High
T7	Deletion of running image	Medium	High
T8	Device cloning	Medium	Medium
T9	Invalid size	Medium	Medium
T10	Higher version	Medium	Medium
T11	Provisioning Server redirection	Medium	Low

Table 4.3: Risk model

include other information to the update image, such as actions to be performed on the update.

Security Requirement 4: Updates failure delay.

The compromise of the provisioning server cannot be detected by the device. The only way to prevent a DoS attack is by delaying the update image reception in case of failure in the verification phase.

Security Requirement 5: Memory object protection.

The protection of memory objects is needed since an attacker could modify, rewrite or invalidate one of them. This can be accomplished by using specific hardware facilities able to protect the content of a Flash memory. In this way, only a specific module in charge of applying the update can modify them, reducing the attack exposure.

Security Requirement 6: Device specific update.

Making each update specific for the device and the request makes sure that is not possible to send old valid updates to a device with a lower version installed. The device should generate a nonce and send it to the server with its unique device id. In this way, each update will contain a specific signature and invalidate all the old update not generated for that device. The robustness of this approach strictly depends on the robustness of the random number generator, on the size of the nonce, and on the effort spent by the attacker to enumerate all the possible nonce for a specific device and version.

Security Requirement 7: Encrypted Request.

Sending the request encrypted prevents eavesdropping attacks from sending invalid data and forcing the device to perform an update. The encryption should be performed using strong and standardized algorithms. The client must always validate the server certificate.

Security Requirement 8: Client Authentication.

To prevent device cloning, each device can be shipped with a unique certificate that is used to authenticate with the server. In case HSM or TPM is used to safely store a secret key, the HMAC algorithm can be used to authenticate the client, considering that is not possible to obtain the key once stored on an HSM.

Security Requirement 9: Safe-Key storage.

When using public key cryptography, the storage of the key can be a problem, since an attacker may be able to modify it, exploiting a software vulnerability and changing the key to invalidate the signature verification. In case an high level of security is required, the public key can be stored on a hardware module such as a HSM and prevent any malicious or random modification.

Security Requirement 10: Use end-to-end encryption

End-to-end encryption between the provisioning server and the update agent makes sure that no other entry points are available on our solution. In fact, it would be possible to use a more powerful edge node to implement security features, but this would increase the attack surface of the final solution.

Security Requirement 11: Device manifest validation

Before the validation of the new update, the values contained in the manifest, including the version, the platform, the size, should be validated. For example, when considering the version the validation means that it must be always higher than the one currently installed. When considering the platform, it must be always compatible with the one already running.

Summary

The previously described requirements must be implemented to solve the identified threats. A correlation between requirement and threats is shown in Table 4.4, indicating which threats are mitigated by each requirement. Moreover, in the last line, the mitigation level is summarized, indicating with **N** the non mitigated threats, with **P** the partially mitigated threats, and with **F** the fully mitigated ones.

The second threat has not been fully mitigated, since we did not explicitly target physical attacks in our architecture. This would require to use a TPM and provide remote attestation to the server. With our solution, even when using an HSM to store the keys, an attacker sufficiently motivated would be able to modify the output of the validation of the HSM using a physical attack, and thus load code on the device that has an invalid signature.

Code	Requirement	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
SR1	Double server architecture			X	X							
SR2	Public-Key based DSA	X					X					
SR3	Double server signature			X	X							
SR4	Updates failure delay											
SR5	Memory object protection		X					X		X	X	
SR6	Device specific update					X						
SR7	Encrypted request						X			X		X
SR8	Client authentication											
SR9	Safe-Key storage	X	X	X	X				X			
SR10	End-to-end encryption								X			X
SR11	Device manifest validation						X			X	X	
Mitigation level		F	P	F	F	F	F	F	F	F	F	F

Table 4.4: Relation between requirements and threats.

4.2.3 Security Configurations

Some of the defined threats do not represent a risk for every possible application. In fact, implementing security is always a compromise between the costs of implementing the security features and the possible future costs involved by not implementing them and, for this reason, some of the previously defined threats may be necessary for some applications.

To reach the needs of each application, we identified four packages with the aim to cover the different security needs when performing an update. The packages are represented in Table 4.5, indicating exactly the security property added by each package and requirements that must be implemented to grant it.

- **Package 1.** This represents the minimum set of features that must be implemented to note that all the configurations include the digital signature verification as a minimum requirement for each update system.
- **Package 2.** The configuration 2 adds the encrypted transmission to protect the traffic from an eavesdropper.
- **Package 3.** The configuration 3 includes mutual authentication, forcing the server to authenticate the client before sending the updates. In this way, unless the device is compromised, no one can access the device firmware.
- **Package 4.** Configuration 4 includes the use of a Hardware Security Module to protect from software attacks that could change the keys stored in the memory of the device and used for validation.

	Integrity and Authenticity	Confidentiality	Client Authentication	Software Attacks Key Protection
Package 1	yes	no	no	no
Package 2	yes	yes	no	no
Pacakge 3	yes	yes	yes	no
Package 4	yes	yes	yes	yes

Table 4.5: Security configurations of the update system.

4.3 Software Updates for Constrained Devices

In the previous sections we analyzed the security threats and derived from them a set of security requirements. However, the identified threads could be generic to every update system and have been already mitigated in many solutions discussed in Chapter 3. In fact, the problem of the available solutions is that they are not suitable for Class 1 devices, suggesting the need of a new solution that merges the security requirements of the update process with the constraints of these devices.

In this section we discuss the requirements for Class 1 devices dividing them into two groups, the *constrained devices requirements*, used to solve hardware and software challenges related to Class 1 devices, and the *portability requirements*, necessary to make the final solution suitable for many application and do not restrict its applicability in case different network protocols or operating system.

4.3.1 Portability Requirements

Portability can be considered as the possibility to reuse the same software in many environments. This requires designing the solution with the appropriate abstraction layer that allows to configure it according to the device capabilities and required logic. We identified four portability requirements:

PR1 Operating System agnostic: Class 1 device can run many operating system, such as the RTOS described in Subsection 2.1.2. However, the described operating systems do not have a standardized API, such an in the POSIX environment. It is required to build a solution that does not explicitly use any OS-specific API but instead relies on abstraction layers that can be implemented for many OS, including RTOS and Linux.

PR2 Netowrk protocol agnostic: The solution must be designed to support many network protocols. For example, in case of constrained devices, it could support CoAP or XMPP and, with more powerful devices, the standard HTTP.

PR3 Cryptographic library agnostic. Many of the security requirements previously described needs to include a cryptographic library into the solution.

This should be done considering that Class 1 devices are very constrained in term of memory and that the application and the update agent should rely on the same cryptographic implementation to reduce the codebase. The solution must be designed with the possibility to use different cryptographic libraries, thus possibly supporting the one included by the user.

PR4 Manifest encoding agnostic. The manifest describing the update can be encoded using different formats, such as simple binary format, JSON, CBOR, and many others, according to the platform needs.

4.3.2 Constrained Devices Requirements

Constrained devices are composed of many hardware types and Class 1 devices correspond to a distinguishable cluster of commercially available chips and design cores [6]. Designing a security solution for these devices needs to take care of many constraints, that will be formalized as requirements in the next subsections.

CR1 Low energy consumption

Class 1 devices are very limited in term of available energy quantity and power sources. The RFC-7228 [6] classification in terms of energy limitation and power source is presented in Table 4.6.

Name	Type of energy limitation	Example Power Source
E0	Event energy-limited	Event-based harvesting
E1	Period energy-limited	Rechargeable battery
E2	Lifetime energy-limited	Non-replaceable primary battery
E9	No direct energy limitations	Mains powered

Table 4.6: Constrained devices energy classification.

Our solution focuses on E1 devices, represented by devices with a rechargeable battery and a limited amount of available energy. This characteristic requires creating an update process that limits the energy usage to avoid depleting the batteries.

CR2 Small memory footprint

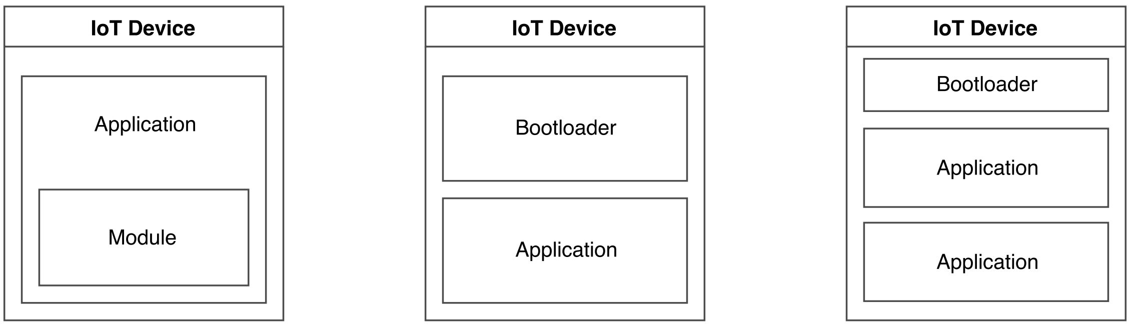
The memory of Class 1 devices is around 10 KiB for the data size and around 100 KiB code size. The small size, especially in terms of data size, implies that every choice in the design must be made to reduce the codebase, such as protocol implementation, cryptographic libraries, structure of the abstraction layer, encoding of the manifest, capabilities of the bootloader, and so on.

CR3 Support various memory types

According to the board capabilities, the device can have one or more accessible memories. We identify as internal memory the one used by the device to run the first application (that can be the bootloader or directly an image). We also consider the possibility of having one or more external memories that in the scope of the update process can store one or more memory objects. The solution must be able to work with memory object located in every type of memory interact with them indiscriminately, allowing to download, manage or remove data from each memory object, for example moving an image from an external memory to the internal memory.

CR4 Support Static, Dynamic, or Seamless Software Updates

According to the capabilities of the operating system, we can have a dynamic, static, or seamless software updates. The solution must be designed to support all of them.



(a) Dynamic. (b) Static. (c) Seamless.

Figure 4.3: Static, dynamic, and seamless software updates.

- *Dynamic Software Updates.* In this configuration, the update image is represented by a module that can be loaded at runtime by the running image [24]. The advantage of this configuration is that no-reboot is needed, making it suitable also for real-time application with high availability needs. To allow the use of this configuration, the OS must be capable of loading, and if necessary linking, the modules at runtime. We will not explicitly manage the activation and relinking of the code, in charge of the OS.

- *Static Software Updates.* In this configuration, the update image is represented by the whole OS. The loading of the update image is not performed by the running image, but the presence of a bootloader is required. The advantage of this configuration is the possibility to perform atomical updates, loading a new image, and avoiding the problems of dynamic linking. Moreover, this requires the reboot of the device that in many applications is not always possible.

- *Seamless Software Updates.* Also known as A/B updates, seamless updates use one memory object to store the running object and another to store the update. All the logic to perform the update is placed in the image and the bootloader needs just to load the newer version, thus each boot will be performed at the same time. This configuration requires that the two memory objects are bootable and thus stored in the internal memory [36]. Considering the size of Class 1 devices memory, this solution is not always usable since the internal memory does not have enough storage for two images. However, considering that the library may be used on less constrained platforms, this approach must be considered during the design phase.

4.4 Software Updates Lifecycle

In this section we analyze the update lifecycle, describing how an update moves during the four phases of the update process and placing together all the components and concepts previously described. We can decompose the update process in four phases: the generation phase, the propagation phase, the verification phase and the activation phase. Figure 4.4 presents an overview of the actions performed by libpull in each phase. The figure represent the Package 4, that as discussed in Subsection 4.2.3, includes all the security features of the solution.

The previous image does not describe all the components of libpull, which will be explained in detail in Section 5.2. Moreover, it represents in a single diagram all the entities involved in the update process, such as the vendor server in the generation phase, the provisioning server in the propagation phase, the HSM in the verification, and a loading module in the activation phase. In this specific case, the latter is the bootloader, but can also be the application or any other module chosen by the library user according to the type of software updates he or she wants to support.

4.4.1 Generation Phase

The generation phase is the first stage in the creation of an update. It does not involve the update agent directly, but mainly the vendor server. It can be considered as the final phase of the developing process when the code is shaped to become an update image. The output of the developing toolchain is an image, a bunch of bytes composed of compiled code that can run on the device and all the required assets needed by the code itself.

The image is not sufficient to perform an update. In fact, it needs to include the manifest, a parsable data describing the content of the image. According to the

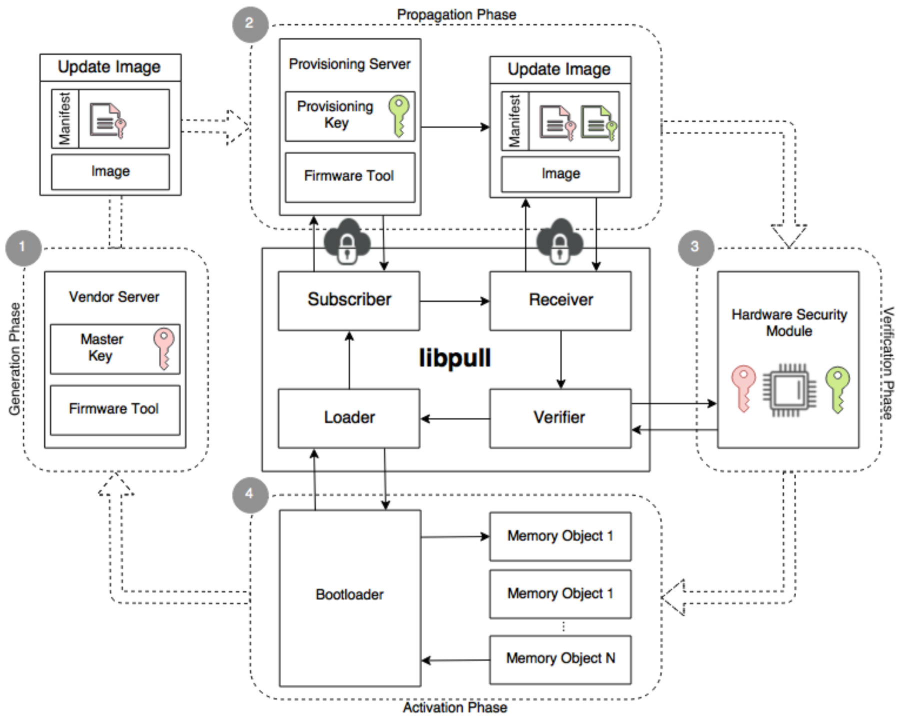


Figure 4.4: Phases of the update process.

requirements, the manifest must contain the following fields:

- The version, since it represents the standard way to compare two images and decide whether the update should be performed or not;
- The size, useful to dynamically handle images of different sizes;
- The application type, useful to differentiates one application from another. In fact, the vendor may use the same device with different application type or configurations;
- The supported platform, useful to differentiate the code of the same application for different platforms. In fact, the vendor may use the same application on different devices.

The manifest must also contain all the required data to grant the integrity and the authenticity of the image and of the manifest itself, such as the signature performed by the two entities, the vendor and the provisioning server, and the hash of the image. The hash of the image must be calculated by the vendor and included in its signature. The provisioning server instead, must add the information on the identity of the device and sign the whole manifest, including also the data generated by the vendor.

The manifest should be designed also to include other data to increase the flexibility of the solution. In fact, the application itself or the loader, a component in charge of loading the update image, may need additional information to handle the process.

During the generation phase is crucial to avoid errors when building the firmware, such as building for a different architecture or not including all resources, since it could make the device inoperable. In August 2017 this happened with hundreds of smart locks produced by a *LockState*, a Colorado-based company. The company shipped faulty software updates that caused a fatal system error [37].

The generated update image must be then sent to the provisioning server, using an out-of-band channel, considering that the vendor server must be always disconnected from the Internet and protected to ensure the secrecy of the vendor private key.

The solution, libpull, is not directly used in this phase. However, since the generated manifest should be compatible with the solution, an utility program has been made able to calculate the digital signature and generate manifest files compatible with the libpull library.

4.4.2 Propagation Phase

The propagation phase consists in the transmission of the update image to the device. The actors involved are the provisioning server and the device itself, communicating with the server using the libpull library. The role of the provisioning server is to ensure the freshness of the update, as identified in **SR6**. We can decompose the propagation phase in two main sub-phases, the subscription phase and the transmission phase.

- **Subscription Phase**

  During the subscription phase, the device interact with the provisioning server to check the presence of an update. The device can be informed of the presence of a new update in two ways: using a polling approach, periodically requesting the latest version available to the server, or using a publish/subscribe approach, in which the device subscribes to a notification channel and gets information when a new update is published.

  When the subscriber module of libpull receives the version, it compares it with the latest one available in memory to checks if it needs to start the update. In this way, the server do not store any data on the device status and can be implemented with a stateless approach, reducing its complexity.

  The data transmitted during this phase can become the target of an attack. In fact, an attacker able to eavesdrop the network can send a new version at each request and force the device to start the transmission phase. This will not lead to the compromise of the device since before applying the update the device always validates the signature. However, the continuous transmission of the firmware could be used to perform a DoS attack and deplete the device batteries. To avoid this problem the connection between the provisioning server and the client should be encrypted and authenticated.

In case a newest version is present, the device can start the transmission phase.

- **Transmission Phase.**

  During the transmission phase, the device receives from the server the update image and stores it in a memory object. It can reuse the connection or start a new one, according to the implementation of the update agent.

  To start the transmission phase, the personalizing of the manifest should be performed, as identified in SR6. The client sends a request to the provisioning server attaching its device id and a random nonce, that will be inserted by the provisioning server into the manifest and signed with the provisioning server key.

  The transmission should be performed in chunks, allowing a fast recovering in case of network failures and thus reducing the energy consumption of the solution when deployed on unreliable networks.

  For the transmission phase, confidentiality is needed for all the applications, where the intellectual property of the image must be preserved. In fact, the technologies included into an image represent for many companies a huge competitive value to be preserved over time. This implies that an eavesdropper able to intercept the communication should not be able to obtain the image. This property can be satisfied using an encrypted transmission and will be satisfied in the final implementation including DTLS.

  Authentication ensures that the device is receiving the data from the correct provisioning server. This implies the need for the update client to authenticate the server. Moreover, in case the protection of the intellectual property of the firmware is required, the authentication of the client is also needed. A common practice for authentication is the use of a username and password, sent from the client to the server. A better practice is the use of a client certificate during the handshake of the secure network protocol.

4.4.3 Verification Phase

The verification phase is in charge of verifying the integrity and authenticity of the update image. This is performed using the two digital signatures contained in the manifest.

The first verification must be performed on the vendor signature. To perform the verification, the same digest operation should be made taking as input the image and comparing the hash with the one contained in the manifest. Then, using the vendor public key, the device can verify the vendor signature validating the hash of the vendor manifest. The public key can be stored in a read-only sector of the memory or in the hardware security module, as shown in Figure 4.4.

In case the first verification succeeded, the verification of the provisioning signature can be done. The public key of the provisioning server can be stored in device memory, on the hardware security module. It can also be stored in the manifest by the vendor server. This allows the vendor to change the public key in case the provisioning server private key has been compromised, implementing a simple key revocation.

4.4.4 Activation Phase

The activation phase is performed on the client and does not involve the servers. It can be performed in different ways and libpull does not force any specific approach.

We describe the static loading since is the one that we used during the evaluation of the library. When the signature has been verified, the device should be rebooted. This may not happen immediately and could allow a malicious attacker to modify the content of the update image during this time window. This implies that, after the reboot, the bootloader should perform again the verification process. In the architecture described in Figure 4.4, the device has many memory objects. This allows to store more than one update image and rollback in case one image is compromised. The bootloader should check the version of each memory object to find the most recent one. It should then compare it with the running image to check if it is higher. In that case, it should validate the signatures and, if both are valid, move the update image on the running object.

The boot process should be as fast as possible, as in this phase the device is not available. The time a device could stay unavailable depends on the field of application. It could be a medical device that should stay on 24/7, or a smart sensor for home automation that could stay unavailable during the night. To decrease the time the device remains unavailable, the bootloader should not perform heavy operations that should be done by the update agent itself. The memory positions in which the new firmware is stored should be structured to avoid unnecessary operations. With device having enough memory, the seamless software update should be preferred, since it reduces the time the device is unavailable requiring the bootloader to only validate and boot the image, without copying the update image to the bootable memory object [4].

Chapter 5

Design And Implementation

Starting from the requirements identified in Chapter 4, it is possible to design and implement the solution, libpull. As already introduced, we decided to make a library that exposes all the functions needed to perform securely a software update and that can be used to create an update agent and a bootloader that better fits the platform requirements. To perform the evaluation, an update agent and a bootloader have been made, but they are not included in the library to keep it as portable as possible, without including any platform-specific code.

The library has been designed as OS-agnostic, able to be used with any OS or even when an OS is not available. During the implementation, it has been tested on Linux and Contiki-NG to evaluate the result, but without including platform-specific code in the library to reduce its complexity and the efforts to maintain it in the future. Every function that required some platform specific functions was defined as an interface, that was implemented in the scope of this thesis for Linux and Contiki-NG, but can be implemented by the library users to support different platforms. This approach of testing the library for Contiki-NG and Linux at the same time was beneficial for the final solution, since they have a completely different approach and execution path. In fact, the execution path of Contiki, based on protothreads, required to structure every module in a way that it does not define any variable internally but everything must be passed in the function signature. Moreover, the goal of the library to support small constrained devices required to avoid dynamic memory, using only static memory for the library and the dependencies. For this reason, the functions that need to work with unknown objects and would require to allocate memory, take as input parameter a pointer to a preallocated object of that type.

In this chapter, we will analyze the library architecture in detail, describing its modules and the interfaces that must be implemented for each platform. We describe also the logic of some application using the library, such as the update agent, the bootloader, and an utility program. Finally, we analyze the testing approach and the testing server developed to evaluate the solution.

5.1 Requirements Summary

We report all the previously identified requirements in a single page, to have a complete overview of the all the abstractions and features the library needs to have. The following requirements will be recalled in the next sections using the defined label.

Security Requirements

SR1: Double server architecture.

SR2: Public-Key based digital signature.

SR3: Double server signature.

SR4: Updates failure delay.

SR5: Memory object protection.

SR6: Device specific update.

SR7: Encrypted request.

SR8: Client authentication.

SR9: Safe key storage.

SR10: End-to-end encryption.

SR11: Device manifest validation.

Portability Requirements

PR1: Operating system agnostic.

PR2: Network protocol agnostic.

PR3: Cryptographic library agnostic.

PR4: Manifest encoding agnostic.

Constrained Devices Requirements

CR1: Low energy consumption.

CR2: Small memory footprint.

CR3: Support various memory types.

CR4: Support static, dynamic or seamless software updates.

5.2 Library Architecture

The libpull library has been designed in different modules and interfaces. The interface are platform dependent and must be implemented according to the platform needs. The modules are based on top of the interfaces and never use platform dependent code. This approach was needed to allow a good portability between different IoT devices. Figure 5.1 visually shows the software architecture modules and their hierarchy.

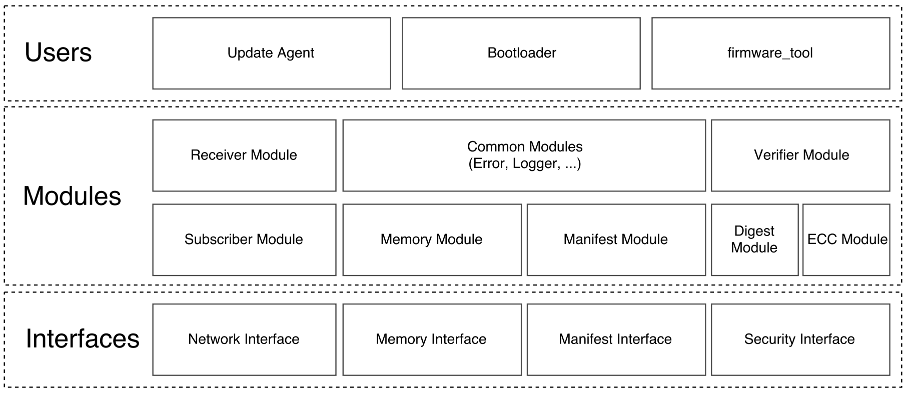


Figure 5.1: Library architecture.

The basic interfaces are:

- **Network Interface.** Used to connect, send, or receive data to an endpoint;
- **Memory Interface.** Used to open, read, write the memory;
- **Manifest Interface.** Implements the manifest using different encoding.
- **Security Interface.** Wrapper for cryptographic libraries.

The modules are based on top of the previously defined interfaces are:

- **Subscriber Module.** Checks the presence of an update;
- **Receiver Module.** Receives and stores an update;
- **Memory Module.** Moves update images from slots and retrieves manifest;
- **Manifest Module.** Perform common operations to the manifest;
- **ECC, Digest, and Verifier Modules.** Calculates digest and verifies digital signature of a memory object.
- **Common Modules.** Used for error reporting, logging, and more.

All these modules can be used by the library users, that we identify as the update agent and the bootloader. We will now analyze each module and interface in detail.

5.2.1 Network Modules

The network modules are in charge of managing the communication between the update agent and the provisioning server. The library requirement **PR2** implies that it should be compatible with any protocol. In fact, libpull does not include any network interface implementation itself, that must be implemented and included for each the specific architecture. However, to evaluate the solution, two testing implementations have been implemented, one for Contiki-NG and the other for Linux.

Network Interface

The network interface must be implemented by the user according to the network protocol he/she wants to use. It is designed as an asynchronous protocol, where is possible to set a callback, perform a request, and wait for the callback to be called. However, the callback cannot be set for each specific request, but only once for the application. It is, however, possible to reset the callback, but all the response will be passed to the single callback. This makes this module not thread safe. However, this limitation is not important, considering that the library is not designed to be used in a concurrent way.

The state of the network is maintained by a network context. This must be implemented according to the protocol, making it an obscure object for the whole library. This object cannot be instantiated by the other modules and must be allocated in the update agent, where the internal structure of the object is known.

A network interface context must be initialized with the `network_init` function, passing the address of the provisioning server, the port on which the service is available, the type of connection and another raw pointer that can contain specific data used by the network implementation. The connection type is an enumeration and is not defined by the library, but it is required since the same network protocol could be used with different configurations. For example, the HTTP protocol could be used on top of TCP or TLS and the CoAP protocol could be used on top of UDP or DTLS. In some specific cases, such as secure TLS of DTLS connections, more data is needed to initialize the network object and can be passed with the raw pointer of the `network_init` function, for example by passing a structure containing the device certificate or a pre-shared key.

The other modules use this interface to perform requests to the end-point defined by the `network_init` function. The request needs to specify a request method, the resource, and send the data and its length. When a response is available, the network interface implementation should call the defined callback with the appropriate values, such as an error in case of error or the data and its size in case of success. Considering that the callback will be invoked on a different time from when it has been set, the ownership of the callback memory must be managed to make sure it is always available.

When the connection to the end-point is no longer necessary, it should be released all the resources and close the connection with the server.

The network interface implementation should be able to open a secure channel to the server, to satisfy **SR7** and **SR10**. This can be done according to the protocols supported by the network interface implementation. In the next subsection, we will analyze how the secure network interface has been implemented for Contiki-NG and Linux.

Contiki-NG Secure Network Implementation

The network interface implementation for Contiki-NG is based on the CoAP implementation included in Contiki-NG itself. This implementation allows to communicate with a CoAP server and is fully integrated with the protothread concept of Contiki, being able to yield the current process until a network response is received or the timeout is exceeded.

The Contiki-NG CoAP implementation is able to manage the Block-Wise connection itself without the need for the programmer to get any block. This feature could become easily a threat if the server is not responding in the correct way. For this reason, the Contiki-NG CoAP implementation was slightly modified to allow the interruption of the Block-Wise reception in case of errors.

The CoAP implementation included in Contiki-NG is implemented on top of UDP and does include natively the possibility to instantiate a secure communication with the server. We modified the CoAP implementation to include the DTLS support based on the TinyDTLS library. The integration required some effort, but it allowed to instantiate a secure connection using the Pre Shared Key (PSK) or the ECC based Diffie-Hellman key exchange (ECDH). Accordingly, the supported cipher suites are:

- `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`;
- `TLS_PSK_WITH_AES_128_CCM_8`.

Linux Secure Network Implementation

To implement the network interface on Linux we used the libcoap library [38]. This library is a C implementation of the CoAP protocol specifically designed for devices with constrained resources. It is compatible with POSIX compliant systems and can be used directly on the device or to perform testing on a standard computer.

The libcoap library supports the Block-Wise transfer. However, differently, from the Contiki-NG implementation, the reception must be handled by the programmer. We found this approach more reliable, as it allows more control over the Block-Wise transfer and the possibility to block the transfer in case of errors.

Our network interface implementation is able to resolve the address using the POSIX standard function, `getaddrinfo`. This allows to pass the domain name of the provisioning server, or to pass directly the IPv6 address.

The library natively supports transport layer security using OpenSSL or Tiny-DTLS. We configured the library to use the latter, for having a similar configuration with the Contiki-NG implementation.

The available cipher suites are `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` or the `TLS_PSK_WITH_AES_128_CCM_8`, since the cryptographic library used is the same for both implementations.

Subscriber

The subscriber module is used to check the presence of an update. The state of the subscriber is stored in the subscriber context, a structure defined by the library itself. This structure is initialized by the initialization function `subscribe`, that receives an already initialized network interface context, the resource on which we want to subscribe, and a temporary memory object used internally by the subscriber to read the version of the defined memory objects. The latest version can be then compared with the version received from the provisioning server.

The subscriber has a default subscriber callback that is implemented by the library. However, if the user wants to interact with a different provisioning server, the subscriber supports the definition of an external callback that matches the server protocol. When the process is finished, the subscriber should be closed using the unsubscribe functions. This function cleans the subscriber context, but does not close the network interface context, which can in this way be reused by the receiver.

Receiver

The receiver module is in charge of receiving the update image from the provisioning server. The receiver module has its own context, used to store information on the received image and on the status of the download. The receiver context must be initialized with an already connected network interface context, the resource from which we want to receive the firmware, and an already open memory object used to store the update image.

The receiver works in an iterative way, exposing the `receiver_chunk` function that receives and stores a chunk of the update image at each iteration. This allows to save internally the reached state and recover from them in case of network failures.

When the image has been received, the `receiver_chunk` sets a flag in the receiver context that must be read by the update agent to stop the execution and start with the update image validation. When the receiver is closed, it does not closes the network context, since if the received object is invalid the network context may be reused again by the subscriber.

5.2.2 Memory Modules

This module is in charge of managing the memory in a more portable way, satisfying requirements SR5, CR3, and CR4. It contains an interface and two modules based on it.

Memory Interface

The goal of the memory interface is to abstract different types of memory that is possible to find on IoT devices, such as raw Flash access or Linux files. Moreover, another goal was to support many memory objects and to be agnostic on their real implementation, hence enabling, for example, to have an object in the internal Flash and another object in the external Flash, and to interact with them using just a single interface. To accomplish this goal, each memory object is represented by a number defined by the library user. The library does not know what is associated with each number and passes it to the memory implementation that maps it internally to the specific type of object. The user must define in its program a global vector of integers called `memory_objects` and declare for each element of the array a number between 0 and 127, that represent the `obj_id`. The array must be terminated with the `OBJ_END` value, defined as -1. This vector is used by the library to iterate on the memory objects, until the `OBJ_END` value is reached.

The interface exposes five functions, used to open, read, write, flush and close a memory object. The flush function has been introduced since the implementation can be done using a buffered I/O. This is especially useful during the write operation, where the flash memory requires to write a full page to perform a correct write, or when the sector must be first erased to be written correctly. In this case, the memory implementation can store the received bytes in the buffer and flush it just when it is full.

Next, we will analyze how the memory interface has been implemented for Contiki-NG and Linux:

Linux: in this implementation, each `obj_id` is represented by a file and the previously described functions are mapped to the standard C I/O functions. Since the read and the write functions allow to receive a relative offset, the function `lseek` has been used to move the file index. Considering that in the Linux implementation the I/O is not buffered, the flush function is not implemented.

Contiki-NG: this implementation is not usable on every Contiki-NG platform since the used functions are specific for the TI cc26xx platform. The implementation maps each `obj_id` with two structures. The first, `memory_object_mapper`, is used to map the ID with the type of flash, in this case, internal or external. The other structure is used to map the `obj_id` to a particular offset of the defined Flash. In this way is possible to define the size of each slot and satisfy **SR5**, checking for invalid access to other objects. Considering that, at this level of the architecture, each slot does not necessarily need to have a manifest associated, we also included a raw object, used by the bootloader interact with the `bootloader_ctx`. The open function takes an `obj_id` and returns a `mem_object` structure, which in this implementation contains the previously described values, such as the type of Flash and the initial offset. This function is also in charge of initializing each Flash the first time it is opened. The read operation uses the `ext_flash_read` function to read from the external Flash or copies directly the bytes in case of internal Flash with direct access. The write function, instead, performs a buffered write, erasing and writing a page just when it is full or when an explicit flush is required. However, due to the RAM constraints, only one write at a time can be performed, since the RAM of Class 1 devices has not enough space to allocate one buffer for each object.

Memory Object

The Memory Object module uses the previously defined interface to provide more high-level functions to the library. Instead of exposing functions such as read or write the memory, it allows analyzing the various memory objects to get the oldest or newest firmware, copy the data from one memory object to another, read or write the manifest stored in a memory object. These actions are necessary for the other modules, but also to the final user, for example, to get the memory object containing the oldest version to start a new update download on that object.

Some of the functions of this module take in input a temporary memory object, used to open internally the various defined objects. Considering that the memory object structure can be defined by the library user, the library is not aware of the content of a memory object and cannot allocate it internally. For example, the `get_oldest_firmware` function, needs to open each slot and to read its metadata in order to to compare the various versions. Thus, it needs a pointer to a memory object structure where the object can be opened.

Manifest

The manifest can be seen as a set of data regarding the image. It can contain arbitrary data used by the final application, but its primary goal is to provide the necessary information to allow the correct version comparison, platform matching, signature verification and to successfully load the update image. Each image that must be transmitted for an update must contain a manifest. Its content is mainly based on the security requirements identified in Subsection 4.2.2.

Considering our architecture based on two servers, we have a general manifest containing two manifests, one filled by the vendor (called `vendor_manifest`) and the other filled by the provisioning server (called `server_manifest`). Those two manifest are contained in a general manifests structure that also includes a signature for the `vendor_manifest` and a signature for the `server_manifest`.

The vendor manifest contains the following fields:

- **size**: the size of the image;
- **offset**: the offset from which the image should be loaded;
- **version**: the version of the image;
- **platform**: the platform for which it has been built;
- **digest**: the digest of the image;
- **server key**: the public key of the provisioning server.

The last field is not mandatory since the public key can be also stored in the internal memory of the client. However, storing the public key of the provisioning server in the manifest allows some kind of revocation in case the provisioning server public key has been compromised.

The server manifest, instead, contains:

- **identity**: a structure containing the identity of the client for which the update image has been built;
- **self checking flags**: a set of flags to be used in the future to support committed boot, a feature that allows testing an update and rollback in case it does not pass all the required tests.

Now that we defined the fields required by the manifest, we need to specify its encoding. Many encodings are possible, such as XML or JSON. However, each of them requires a parser and, considering the constraints of our platform and requirement CR2, such a choice would increase the size of the final solution. Hence, for our solution, we decided to encode the manifest in binary format, using a standard C struct. The advantages are obviously the small representation and the possibility to read it without any explicit parser, just using a binary read in C. However, the solution has been designed and implemented to support future encodings and satisfy PR4, being agnostic to the manifest format. During the whole development of the library, much effort has been spent to ensure no direct access to the manifest struct and that every field is accessed using a specific getter method. This makes easier to support in the future other manifest encodings and to change the data organization without changing the code that works with it.

A suitable encoding format for embedded devices is the Concise Binary Object Representation (CBOR), a standard defined by the IETF in RFC 7049 [39]. It is a data serialization format based on JSON and allows to transmit data objects that contain name-value pairs in a binary form. Similar to JSON, it does not require the use of any schema and is widely extensible to integrate other data types. It has been designed to allow extremely small code size, and the smallest implementation of a parser is just 880 bytes. It is the serialization protocol to be used in conjunction with CoAP, thus it would be a good match with our architecture.

5.2.3 Security Modules

The implementation of the security modules consists of high-level interfaces implemented using different cryptographic libraries. This allows to perform the verification of a memory object using the same code, but using different cryptographic libraries, enabling the library users to include the one that better fits their requirements. This design satisfies the **PR3** requirement.

Supported Libraries

Libpull currently supports three cryptographic libraries:

- **TinyDTLS** [40]. It is a library that provides all the functions to create a DTLS connection. It supports many cryptographic algorithms, such as Rijndael (AES), SHA256, HMAC-SHA256, ECC (with secp256r1 key). It can perform the DTLS handshake using PSK or the ECDH algorithm. It is distributed under the MIT license and maintained by the Eclipse for IoT project.
- **TinyCrypt** [41]. It is a small-footprint cryptography library that targets explicitly constrained devices. It supports many cryptographic algorithms, such as SHA-256 hash functions, HMAC-SHA256, AES-128 (with AES-CBC, AES-CTR, and AES-CMAC encryption modes), ECC-DH key changes, and ECDSA. It is built in a modular way, allowing to include only the required modules. The library is maintained by Intel.
- **Atmel CryptoAuthLib** [42]. This library is provided by Atmel to interact with their CryptoAuthentication modules. It is a very modular library and bases its function on a HAL layer in charge of communicating with the device using I2C or SPI. The library exposes a simple API and an advanced API and allows to build custom commands to be sent to the HSM. The library needed some patchs to work correctly with Contiki-NG and our board. We used this library to interact with with the ATECC508a and ATSHA204a chip, connected to the board using the DevPack interface and the I2C bus. The DevPack is shown in Figure 5.2, and has been made by Markus Schuß, a PhD student at Graz University of Technology supervising this thesis.

Figure 5.2: DevPackSecure.

These libraries are included to perform SHA-256 digest and ECDSA verification.

Cryptographic Libraries Memory Footprint

The choice of the cryptographic library to include was sustained by an analysis of the memory footprint of several cryptographic libraries, to identify the smallest in terms of Data and Text size. On the final implementation for Class 1 devices, based on Contiki-NG, the loading of the update image must be done by the bootloader, that must also validate the update image before loading it, including a cryptographic libraries. This implies that the cryptographic library will be included two times in memory, and thus its memory footprint must be as smaller as possible.

The comparison has been performed building a simple application able to perform the verification for each library and comparing the size of the hashing function and the ECC functions. The output of the comparison is presented in Table 5.1, and shows that TinyDTLS is the smaller library for performing verification. However, the memory footprint difference with tinycrypt of 11535 bytes makes them both a good candidate for our implementation. The libraries has been build for the Linux architecture, and evaluated adding the following compiler flags to the build: `-Os -ffunction-sections -fdata-sections -Wl,--gc-sections`.

Library	SHA2	ECC	ECDSA
TinyDTLS	3800	7531	9888
tinycrypt	3656	8968	11241
PolarSSL	6056	23046	27735
MatrixSSL	3864	29103	34022
WolfSSL	4592	31443	34777
LibTomCrypt	4354	35959	38256

Table 5.1: ECDSA memory footprint.

Digest Module

The digest module is used to enable the security functions to use different digest algorithm and implementation library without changing the code. For example, the function used to verify the signature may perform the hashing of the update image using the SHA-256 or the SHA-512 algorithm. Moreover, the previous algorithm could be implemented using different cryptographic libraries, for example *tinycrypt* or *TinyDTLS*.

This modularity is reached using a structure, `digest_func`, that contains function pointers for the specific digest functions, the size of the produced digest and a buffer of that size used to store the result. This implementation is supported by the fact that all the analyzed libraries use the same approach to calculate the digest. The first step is to initialize the hash context, then update the hash with data chunks, finally get the hash with the finalize function. For each algorithm and library, we implemented a function wrapping the real one, such that all of them have the same function signature. Moreover, to simplify the functions and reduce the memory footprint, for each combination of algorithm and library, a new function has been created. This means that the CryptoAuthlib hardware and software digest is implemented with two different functions, but that exposes the same API.

ECC Module

The ECC module is used to expose the Elliptic Curve Cryptography functions in a portable way between the different used implementations. Its implementation is very similar to the already discussed digest implementation, using a struct of function pointers to instruct the functions on the specific algorithm and library to use.

For each library and algorithm, two functions have been created, implementing the high-level ECC interface. The first is the `ecc_verify` function, that takes as input the X and Y parameter of the signer public key, the R and S parameter of the signature, the data to be validated and its length. The type of curve to be used is indicated by the name of the function. For example, in case we want to perform the verification using TinyDTLS and the *secp256r1* curve, we will use the `tinydtls_secp256r1_ecc_verify` function. If, instead we want to use the tinycrypt implementation, the `tinycrypt_secp256r1_ecc_verify` function must be called. The second interface that has been implemented is the `ecc_sign` function, that allows generating a digital signature using the defined algorithm. It follows the logic previously described for `ecc_verify`.

Verifier Module

The verifier module uses the digest interface and the ECC interface to verify the update image signature, working directly with a memory object. It takes as input the already open memory object, the public key X, and Y parameter, and the digest and ECC function struct previously defined.

The function initially performs the digest of the memory object and compares it with the digest stored into the manifest. To calculate the digest it reads the image in chunks into a buffer passed by the caller, that can define the size of the buffer accordingly to the platform constraints. If the digest is not correct the function returns, otherwise, it moves forward verifying the vendor and the server signatures. If both are valid the verification is successful.

This verification function could be improved by splitting the verification function in two distinct functions, one to verify the digest and the other to verify the manifest signature. This would empower the receiver to verify the signature as soon as possible, not receiving the whole update image in case the validation does not passes.

5.2.4 Common Modules

The previously identified components are necessary to perform the update process. However, each component needs to deal with error reporting and logging, to make easier testing and integrating the library. The common modules are included by all the other modules. Some of them are:

- **Error.** Defines an enum containing the errors for each module. It also generates a string for each error, used to log an error in a readable but memory efficient way. This reduces the size of the error logging functions, since it allows to log errors printing only their name without adding specific strings. It could be useful to debug the library with high constraints in term of memory, that prevents to include all the the debugging strings;
- **Logger.** Provides logging function and macros with different logging level, used to debug the library and reduce the logging output if not necessary. Four incremental verbosity levels are supported: error, warning, information, debug. When a verbosity level is selected also all the previous level are enabled. This means that in case the information level is selected, also errors and warnings are printed. The verbosity level can be selected at compile time, defining a preprocessor macro, or at runtime, defining a global symbol. The definition at compile time allows the compiler to remove all the logging instructions with a verbosity level lower than the previously defined one, reducing the memory footprint. When defined at runtime instead, all the logging instructions will be included also if a low verbosity level is used;
- **External.** This module is very important since it defines the global symbols that must be implemented by the user. These global symbols are needed to pass some configuration to the library function, without having a specific argument on each function. Considering that external unresolved symbols can generate linker errors difficult to understand, the only external symbol that is required by the library is a vector of memory object id, that will be used to compare the memory objects to find the oldest or the newest one;
- **Types.** Defines custom C types that are used in the whole library. This module is important to grant consistency, for example defining in one single point the size of the *version* or the size of the pointer used to reference internal memory;
- **Identity.** Defines a structure that is used to declare the device identity, such as a Unique Device Identification (UDID), and pass it to the various functions, such as the receiver that needs to verify the identity contained in the received manifest;
- **Callback.** Defines in one single point the format of the callback used in the library. It is mainly used from the network modules, such as the network interface, the receiver, and the subscriber.

5.3 Library Users

To evaluate the library we created two update agents, one for Contiki-NG and one for Linux, and a bootloader, used on the board running Contiki-NG to apply the updates. We consider all these implementations as users of the library and, in fact, they are not included in the library itself, that aims to be used to build any generic update agent, bootloader or application the user needs. However, they can be used as a guide for implementing other update agents, and for this reason, the design and implementation choices will be discussed in the next sections. Moreover, we will introduce a utility that has been built to support the generation process and that is needed to generate manifest compatible with the libpull library.

5.3.1 Update Agent

The update agent is the application using the libpull library to effectively perform the update. It is in charge of communicating with the network and coordinate the operations to successfully download, verify and apply the update image. It should be normally executed in parallel to the standard application. In this way, when an update is available, the device will require the minimum time to obtain it. However, the update agent could be started only at specific time intervals, according to the device and application requirements.

We developed two update agent. One for Contiki-NG and one for Linux. They have the same logic, but a slightly different implementation was required due to the Contiki-NG execution model, based on protothread, that requires each process yielding its execution to return the control to the main loop. This can be done only from a process itself and thus the Contiki-NG update agent has been implemented as a Contiki-NG `PROCESS`. The Linux version instead has been used as integration test to validate the whole architecture. It downloads the update to a file instead of using the flash and uses the network interface implementation based on *libcoap*.

The Linux implementation can be compiled as a standalone program and used to update a specific file. If more logic is added to the update agent, it could be used to check the update for different files. However, this is not the generic approach for which the library has been developed.

The update agent defines all the configurations of the library, such as the endpoint used, the resources that must be contacted by the subscriber and the receiver, the type of connection that must be used, the polling timeout. Moreover, it must be able to recover the various errors, and fail if they are not recoverable or try to recover them if it is possible. For this reason, the update agent has been implemented as a while loop that exits only if an unrecoverable error has been encountered or the update process is successful.

Figure 5.3 shows the main operations performed by the update agent, and each operation is represented by the effective function exposed in the libpull library.

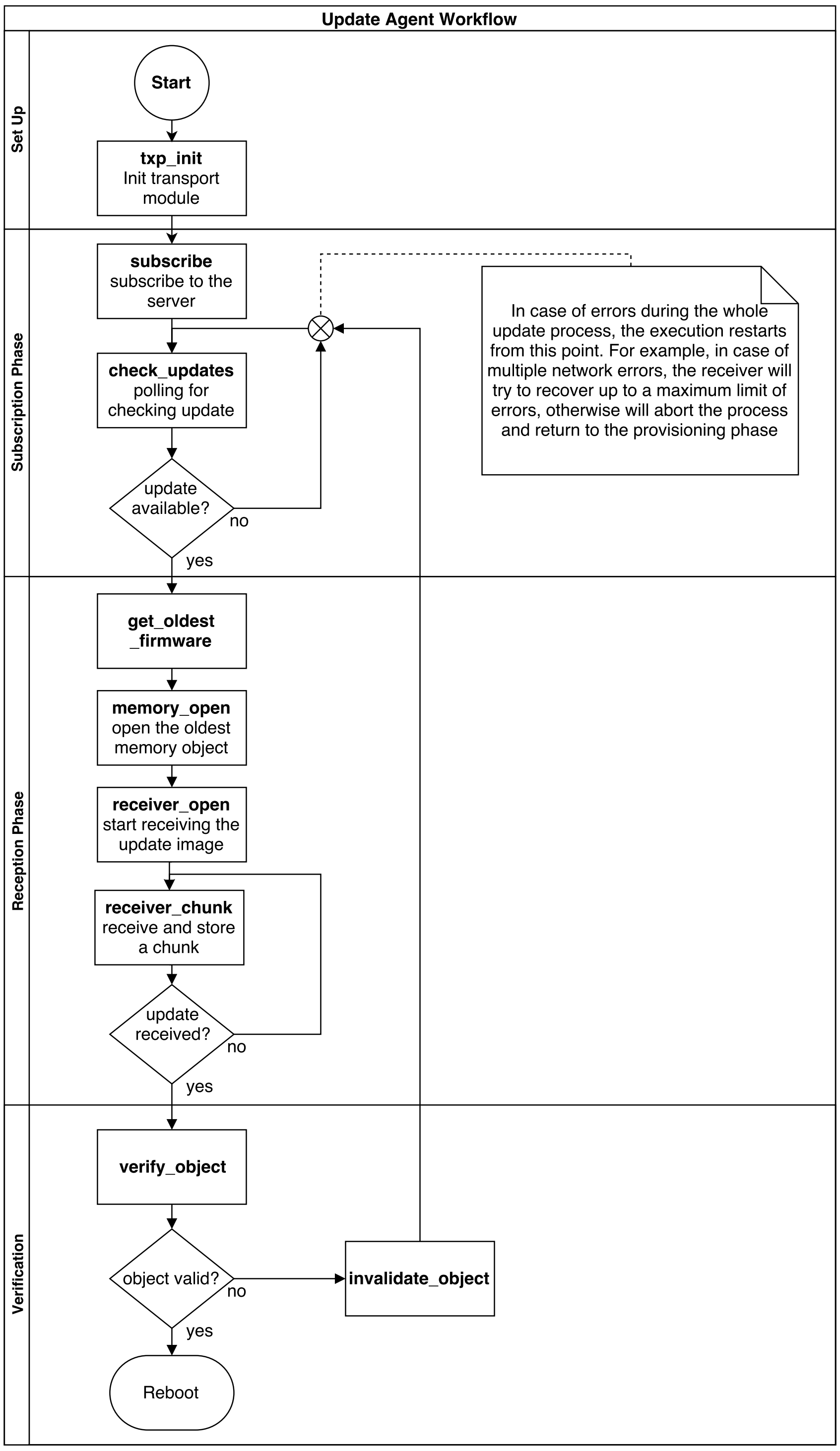

Update Agent Workflow
Set Up
Subscription Phase
Reception Phase
Verification
Start
txp_init
Init transport module
subscribe
subscribe to the server
check_updates
polling for checking update
update available?
no
yes
In case of errors during the whole update process, the execution restarts from this point. For example, in case of multiple network errors, the receiver will try to recover up to a maximum limit of errors, otherwise will abort the process and return to the provisioning phase
get_oldest _firmware
memory_open
open the oldest memory object
receiver_open
start receiving the update image
receiver_chunk
receive and store a chunk
update received?
no
yes
verify_object
object valid?
no
yes
invalidate_object
Reboot

Figure 5.3: Update agent execution flow.

5.3.2 Bootloader

The Contiki-NG implementation was based on static loading, described in Section 4.3. This requires storing the running image in the internal Flash memory and the supplementary memory objects in the external Flash memory. The images are moved from one slot to the other by the bootloader, that must be located at the address where the CPU starts its execution. We see the bootloader as a user of the libpull library and, in fact, has been implemented using the libpull APIs to access the memory and perform validation.

Some parts of the logic and the implementation of the bootloader were inspired by an already developed project for the TI CC2650, developed by Mark Solters [43].

We developed the bootloader on top of a stripped version of Contiki-NG. This approach, compared to the one of building the bootloader as a standalone application, increases the final size of the bootloader but also reduces the complexity to port it to another architecture supported by Contiki-NG. To reduce the size of Contiki-NG it was required to modify some internal functions in charge of loading some board peripherals. This approach also facilitates the interaction with the external memory and the I2C bus, that are initialized during the Contiki-NG startup process for each supported platform.

The execution of the bootloader differs from the first execution, called bootstrap, to the successive executions. During the bootstrap the external memory objects located in the external flash are erased, and if the recovery image is enabled, the running object is stored into a specific memory object located in the external memory, allowing a fast recovery in case of failures. To recognize the first run from the other runs, the bootloader needs to store its state in a persistent memory. This is done defining a new memory object, called `bootloader_ctx`, that is stored in the last page of the internal memory and contains a bit, used to indicate if the bootloader is running for the first time. The `bootloader_ctx` is generated during the building phase and flashed to the board at the correct offset. The `first_run` bit is initially set to one and can be only set to 0 once by the bootloader, since to reset it an erase of the last sector is required, but in the board used for testing is not possible since it contains the CCFG, a structure containing booting parameters read by the CPU at boot time.

If the bootloader finds a newest version in one object of the external memory, it verifies the signature of that object and, if valid, copies it to the running object, ready to boot. We performed the verification of the newest firmware before the copy, directly on the external Flash. This requires more time, since the external flash memory is slower, however if the object signature is invalid it does not make sense to copy the object to the internal Flash memory, that later needs to be restored. Moreover, the internal Flash memory is accessed through a cache and if its content changes the cache must be invalidated and reloaded.

An important operation performed by the bootloader is to write protect all the sectors of the internal Flash before loading the image, blocking in this way the update agent and all the software running after the bootloader from modifying all the content of the internal memory and prevents an attacker to store persistent data in the internal Flash. The execution flow is summarized in Figure 5.4.

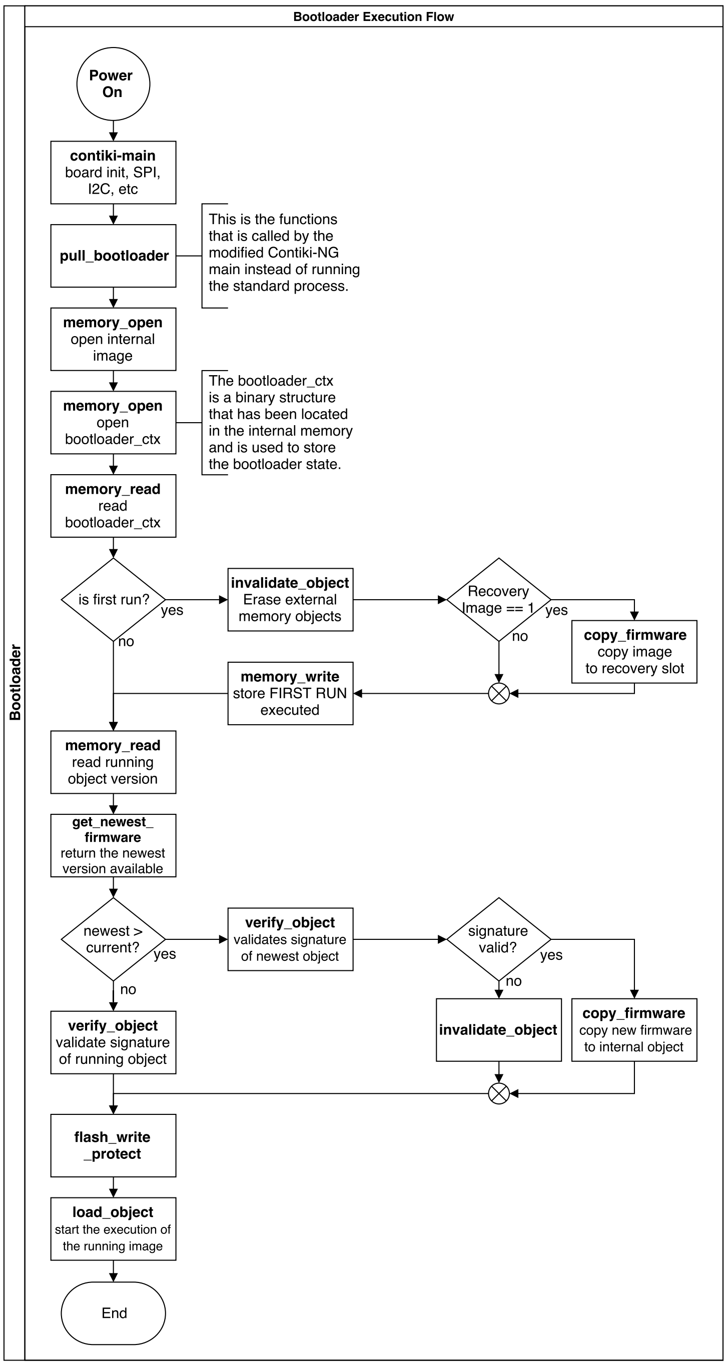

Bootloader Execution Flow
Bootloader
Power On
contiki-main board init, SPI, I2C, etc
pull_bootloader
This is the functions that is called by the modified Contiki-NG main instead of running the standard process.
memory_open open internal image
memory_open open bootloader_ctx
The bootloader_ctx is a binary structure that has been located in the internal memory and is used to store the bootloader state.
memory_read read bootloader_ctx
is first run?
yes
no
invalidate_object Erase external memory objects
Recovery Image == 1
yes
no
copy_firmware copy image to recovery slot
memory_write store FIRST RUN executed
memory_read read running object version
get_newest_firmware return the newest version available
newest > current?
yes
no
verify_object validates signature of newest object
signature valid?
yes
no
invalidate_object
copy_firmware copy new firmware to internal object
verify_object validate signature of running object
flash_write_protect
load_object start the execution of the running image
End

Figure 5.4: Bootloader execution flow.

5.3.3 Firmware Tool

To generate the update image an utility program has been implemented, called `firmware_tool`, able to accomplish many operations related to the creation of the update image. As already introduced an update image includes the image, that is the output of the build toolchain, and the manifest, that contains information able to describe the image itself. The manifest must be created according to the encoding used, that in our case is a simple C struct. The `firmware_tool` utility is able to generate the manifest when invoked with the correct command. The first implementation of this utility program has been done in C, however, it has been moved later to C++ 11, since it enables a higher level programming and includes in the standard library many useful features used to build the tool. The tool can be invoked with three different commands:

- **keys**: allows to generate or validate a set of public keys;
- **manifest**: allows to generate, validate or print the content of a manifest;
- **configs**: allows to validate, print or store the configurations.

The goal of the first and the second command are obvious in the context of the update process. We included the third command following the approach used by many modern tools nowadays, such as using a configuration file in the folder where the program is executed to import a specific configuration, removing the need of passing the arguments in the command line. This means that the `firmware_tool` program is able to take the commands as a standard command line parameter (i.e., -f for the output file, -k for the signing key) but also importing them from a configuration file. This has several advantages, such as:

- have a configuration file for each folder containing a different project, without the need of building complex scripts to invoke the program with the correct arguments;
- allow the versioning of the configuration file with the project itself, knowing exactly the configuration used to build the update image for each specific version;
- allow validating and modifying the configuration before applying them, for example checking the existence of all the required signing keys before invoking the program;
- validating the manifest using the same parameter used to build it, reducing errors and the need of storing them explicitly.

To store the configuration the JSON format was used first. However, after a review of the possible encodings, we decided that TOML[44] was the more appropriate to storing configurations since, in comparison to JSON, has a more understandable encoding, the possibility to include comments into the configuration (i.e., explicitly document a configuration), and an easier syntax compared to JSON and YAML.

When invoked the `firmware_tool` checks the presence of a default `libpull.toml` file in the current directory and, if present, loads from there the parameter. In this way, invoking `firmware_tool manifest generate` is sufficient to generate a valid manifest with all the required fields for that specific update image.

5.4 Testing

Much effort has been spent on testing the implementation. Having a high code coverage level with *Unit Testing* was a requirement to ensure the correct behavior of the single functions that are composed to create the update agent. Moreover, testing them on the developing machine reduces the developing and debugging time compared to a direct testing on the developing board.

To reach a high coverage level, a deep analysis of the behavior of each function needs to be done, going through all the possible execution path and ensuring that a function is able to fail correctly in case the called functions return an error. This implies the need of managing the return values of the functions used internally by the component we want to test. This is normally performed using function mocking, a technique not easy to implement in C, at least without complex algorithm or additional code generation. There are three main approaches to perform function mocking in C:

- **Using weak symbols.** This technique consists in defining the mocking functions as hard symbols during the testing phase and override the weak symbols of the implementation. This approach has been avoided since it can introduce many bugs and result in linker error when building the library with different compilers;

- **Use specific compiler directives.** This technique requires to pass a specific command line argument to the compiler during the test build, indicating the symbol that needs to be replaced. This approach has been discarded since it is not portable to all compilers. For example, the command to replace a symbol is not equal in *gcc* and *clang*.

- **Mocking code generation.** This technique requires to generate, automatically or manually, additional mocking code that must be linked instead of the real implementation to change the behavior of the functions.

The last approach has been used since, even if it requires some more effort, it is the more portable solution and allows to run tests directly on the development board, cross-compiling them with the correct compiler. To generate the mocking code, the *CMock* framework has been used initially. This is a ruby script that parses a C header and automatically generates another C header able to mock the defined functions. This tool was integrated with the *Ceedling* build system, that was initially used as a test runner. This choice was supported by the low effort required to maintain this build system and allowed to reduce the development time during the first phases since it automatically tracks each newly added module. During the last phase of the development, the *Ceedling* build system has been replaced with the standard GNU build system, that using Autoconf and Automake generates a Makefile able to build the library and the tests itself. This allowed removing *Ceedling* and the ruby dependencies in favor of a more standard way of building C libraries. The *CMock* framework used to autogenerate the mocking files was removed with *Ceedling*, and considering that the number of files that needs to be mocked is currently limited, the mocking files have been generated manually, using a struct of function pointer to redirect the function execution.

To perform the tests the *Unity* testing framework has been used. After a review of the possible frameworks, we found on *Unity* the most readable, dynamic but also a simple testing framework, fully compatible with embedded devices. This choice was motivated by the idea that while porting the library to a new platform, the interfaces can be validated running the tests directly on the device.

Testing Server

To test the implementation a testing server has been implemented. The language used for the implementation is C, that allows to include directly the structures used in the library without the need of translation to other languages. The server listens for CoAP and CoAPS request, and uses `libcoap`[38] as a network library. The server exposes various resources, usable to check the version and receive the update. Moreover, it exposes also resources useful for the testing itself, such as an `invalid_version` to test the subscription module, a `next_version` resource that increments the current version and generates a new signature, useful when a sequential download is required, and a `invalid_size` resource used to send an update with an invalid size and test the receiver implementation.

Continuous Integration

The unit and integration tests have been executed using *Travis CI*[45], an online service to perform continuous integration and execute the tests for each new commit performed on a linked repository. This service has been integrated with GitHub, used as a versioning service, and was very useful to detect regression during the developing phase, ensuring that every newly added feature does not compromise the already implemented one. This tool has been integrated with Coveralls[46], an online service that allows to monitor the coverage percentage over time.

Chapter 6

Evaluation

The solution has been evaluated in terms of memory footprint, execution time, and energy consumption. Only the Contiki-NG version has been evaluated since is the only one that we were able to test on a real board. The Linux version has been tested only on the developing machine.

To perform the evaluation we used two TI SimpleLink Sensortag CC2650, described in Subsection 2.1.1. One acting as the device that needs to be updated, and running the bootloader and the update agent, and the other acting as a border router, to interface the 6LoWPAN network with the interface of the computer running the server. We used the ping command to measure the round-trip-time between the computer running the server and the device that needs to be updated, that was of 17.842 ms with a standard deviation of 2.420. To perform the test in a repeatable way a fake update image was created with a size of 91kB. In fact, during the first evaluation, the update agent was recursively downloading itself with a newer version, however, this approach was faulty since the size of the update image is different when using different configurations for network and cryptographic libraries. For this reason, the update agent was slightly modified to download a fake firmware (with a valid manifest) and invalidate it before the activation phase.

To increase the quality of the data, removing outliers and judging the repeteability, we performed each evaluation ten times, to have an average value and remove possible outliers. This process was supported by a set of preprocessor macros created to obtain time and energy consumption in a specific code section, based on the Contiki-NG Rtimer and Energest. For each measurement, the module logged a comma-separated value (CSV) to the serial interface, that was stored in a file for a successive post-processing. All the possible combinations of cryptographic libraries (TinyDTLS, tinycrypt, CryptoAuthlib) and connection (simple UDP, DTLS PSK, DTLS ECDSA) have been considered, but not the combination of DTLS and tinycrypt since the verification primitives are already provided by TinyDTLS used to perform the DTLS connection. The energy consumption has not been measured when using DTLS ECDSA due to an incompatibility with the Energest module.

6.1 Memory Footprint

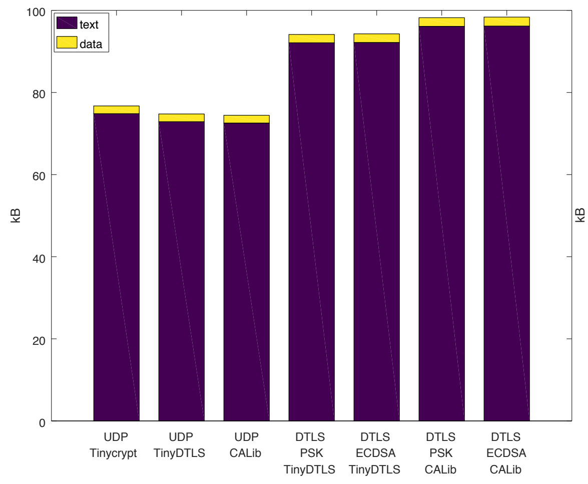
Considering requirement **CR2**, “small memory footprint”, we need to ensure that the library has the smaller memory footprint as possible and that it fits on a real Class 1 device. The ROM and RAM size of the device used for the implementation are respectively of 128 kB and 20 kB.

The library has been built for each configuration of update agent and bootloader and its size has been evaluated using the `size` command line utility, passing in input the Executable and Linkable Format (ELF) file generated by the build system. Since each configuration includes different logging strings, we removed all the logging output of the library before calculating the memory footprint.

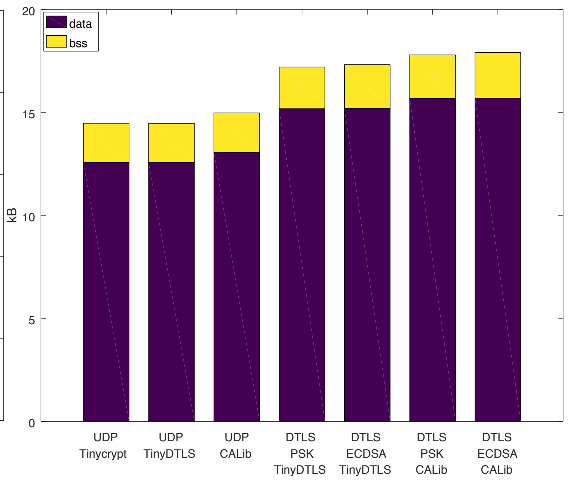
Table 6.1 shows the memory footprint of the update agent for each configuration of connection type and verification library. The size includes the Contiki-NG operating system, the CoAP library and the cryptographic library.

Connection	Verification Lib.	Text	Data	Bss	Tot. RAM	Tot. ROM
UDP	Tinycrypt	74812	1909	12560	14469	76721
UDP	TinyDTLS	72860	1905	12560	14465	74765
UDP	CryptoAuthLib	72532	1908	13064	14972	74440
DTLS_PSK	TinyDTLS	92097	2029	15172	17201	94126
DTLS_ECDSA	TinyDTLS	92147	2129	15188	17317	94276
DTLS_PSK	CryptoAuthLib	96097	2113	15676	17789	98210
DTLS_ECDSA	CryptoAuthLib	96147	2213	15692	17905	98360

Table 6.1: Contiki-NG update agent memory footprint.



(a) Total required ROM.



(b) Total required RAM.

Figure 6.1: Contiki-NG update agent memory footprint.

To optimize the library it was necessary to analyze several times the symbols exported, using the `objdump` command line utility, to optimize the ones with the highest impact in *text*, *data* or *bss*.

As it is possible to see from Table 6.1, the size of the update agent fits perfectly in the available ROM and RAM of the tested device. However, the ROM must also be shared with the bootloader that is placed at offset 0x0 of the internal flash. The size of the bootloader is shown in Table 6.2 and in Figure 6.2. The bootloader was able to fit inside 4 pages of size 0x1000 bytes, for a total size of 16kB. As already explained, the bootloader is based on top of a Contiki-NG image to make it easily compatible with every platform supported by this OS. A bootloader made as a standalone program would have a smaller memory footprint but a higher complexity to integrate it with other platforms.

Bootloader Configuration	Text	Data	Bss	Tot. RAM	Tot. ROM
TinyDTLS	14420	313	5948	6261	14733
Cryptoauthlib	14669	361	6416	6777	15030
TinyCrypt	15765	280	5908	6188	16045

Table 6.2: Contiki-NG bootloader memory footprint.

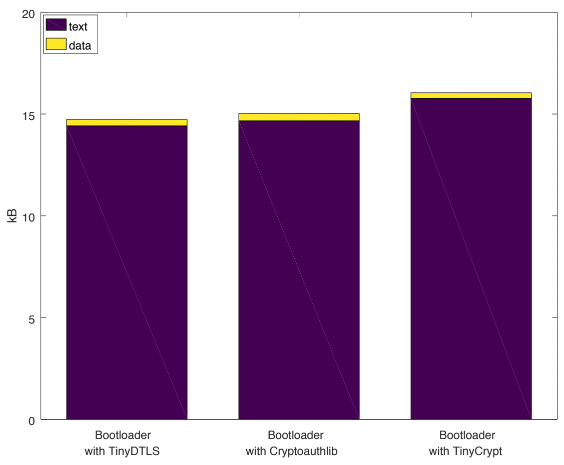


Figure 6.2: Contiki-NG bootloader memory footprint.

The total RAM required has been calculated as the sum of Data and Bss. The first contains all the initialized variables that must be loaded from the binary during its execution, the second contains instead the uninitialized variables for which is only necessary to allocate memory without loading its value from the binary. Consequently, the total ROM usage has been calculated as the sum of Text and Data, since the initialized variables must be stored in the ROM. The Text section contains all the executable code.

6.2 Execution Time

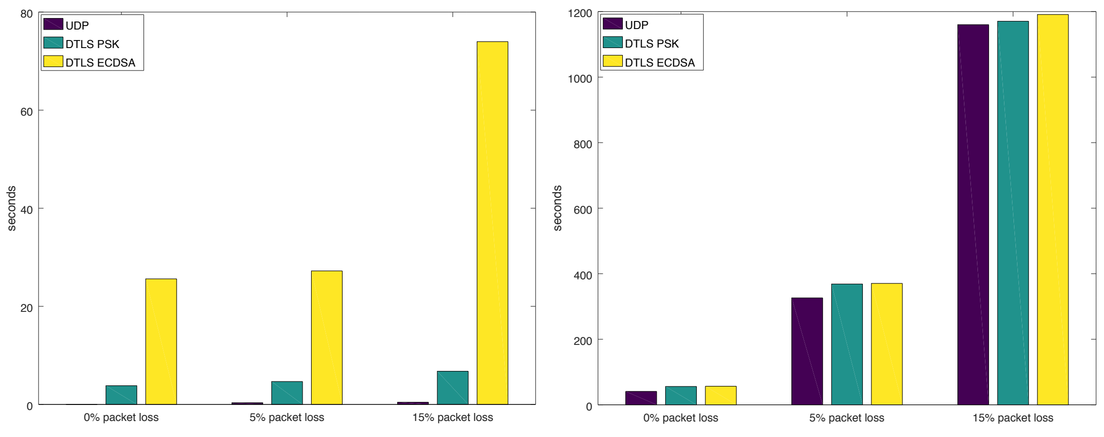
To evaluate the time of each phase, the Contiki-NG *RTIMER* has been used, that provides scheduling and execution of real-time tasks, with predictable execution times. The execution time has been measured for each phase, from the subscription to the verification one.

Since the subscription phase requires to communicate with the network, we modified the update agent to wait for a valid network setup, ensuring that the RPL network has been set up and that a route has been found, able to route 10 ICMPv6 Echo Request packet to the server.

To make the evaluation more reliable and similar to a real environment we used a feature of the testing server to simulate a packet loss. We used three different values of packet loss: 0%, 5% and 15% to simulate the different possible environment.

Network and	Subscription (s)			Reception (s)		
packet loss	Min	Max	Avg	Min	Max	Avg
UDP (0%)	0,0196	0,0200	0,0198	40,1780	44,8667	40,7912
UDP (5%)	0,0197	3,2722	0,3454	290,5921	371,9002	326,2205
UDP (15%)	0,0210	4,2962	0,4502	1.088,0753	1.277,3947	1.160,1717
DTLS PSK (0%)	3,0382	4,4221	3,8186	55,7232	56,2592	55,8622
DTLS PSK (5%)	3,3447	12,4348	4,6692	280,3839	423,6849	368,5568
DTLS PSK (15%)	3,6335	23,2770	6,7628	1.075,7546	1.296,8305	1.170,5456
DTLS ECDSA (0%)	23,9700	26,8803	25,5986	55,7668	59,3859	56,5797
DTLS ECDSA (5%)	24,5148	33,2222	27,2220	325,6834	448,2559	370,4337
DTLS ECDSA (15%)	24,5753	349,7438	73,9708	1.084,1773	1.344,2163	1.191,0994

Table 6.3: Propagation phase execution time.



(a) Subscription phase. (b) Reception phase.

Figure 6.3: Propagation phase execution time.

The packets are discarded in a random way by the server and for its total duration and not the duration of each phase. This means that when the server is configured with 15% of packet loss, there could be a subscription phase with 0% of packet loss and a subscription phase with 50% of packet loss, thus performing multiple measurements was even more important to find an average value similar to a real environment.

As we can see in Figure 6.3b, the reception time is very similar during the reception for each network configuration. This can be explained by the fact that the DTLS handshake is performed only during the subscription phase and the connection is reused also for the reception phase. For this reason, the subscription phase time is much increases from UDP to DTLS PSK and increases significantly from DTLS PSK to DTLS ECDSA, as shown in Figure 6.3a.

As is possible to see in Table 6.3, the packet loss impact is higher in the configuration with DTLS ECDSA, since it requires several packets sent in the correct order to perform a valid handshake. Is possible to notice that instead of that in the reception phase the packet loss percentage has the same impact in all the configurations, moving from a value in the order of 50 seconds with no packet loss, to a value in the order of 20 minutes with 15% of packet loss.

Verification Phase		TinyCrypt	TinyDTLS	CryptoAuthLib
Digest (ms)	Min	455,29	422,58	403,20
	Max	455,35	422,64	403,29
	Avg	455,32	422,60	403,25
Vendor Signature Verification (ms)	Min	567,99	6449,80	86,91
	Max	568,02	6449,83	86,98
	Avg	568,01	6449,82	86,94
Server Signature Verification (ms)	Min	567,47	6245,79	86,79
	Max	567,50	6245,88	91,86
	Avg	567,50	6245,81	87,34

Table 6.4: Verification phase execution time.

Table 6.4 shows, as expected, that the verification time is always constant between the iterations. In fact, it only slightly differs of few milliseconds in the server verification when using CryptoAuthLib and ATECC508a, due to a little response delay of the device when it is switched to sleep mode after the first verification.

In the case of CryptoAuthLib, the digest is calculated using the software implementation provided by the library. Although the ATECC508a is able to calculate the SHA-256 digest, sending 90 kB in chunks of 64 bytes to the device is much slower than using the software implementation. The hardware implementation could increase the security of the solution because it is able to verify a signature performing the digest algorithm internally and using the output as directly for the verification.

However, this path has not been explored considering the higher time required to send all the chunks to the device.

Table 6.4 also shows the higher time required by TinyDTLS to perform an ECDSA verification. This operation must be also performed during DTLS handshake when using the ECDSA configuration and can explain the higher value compared to the DTLS PSK handshake. This makes the configuration with TinyDTLS the only one where the verification has an impact on the time of the update process, as shown in Figure 6.4.

Configuration	Subscribe (s)	Receive (s)	Digest (s)	Vendor Signature (s)	Server Signature (s)	Total Time (s)
UDP CryptoAuthLib	0,0198	40,7912	0,4033	0,0869	0,0873	41,3885
UDP Tinycrypt	0,0198	40,7912	0,4553	0,5680	0,5675	42,4018
UDP TinyDTLS	0,0198	40,7912	0,4226	6,4498	6,2458	53,9292
DTLS PSK CryptoAuthLib	3,8186	55,8622	0,4033	0,0869	0,0873	60,2583
DTLS PSK TinyDTLS	3,8186	55,8622	0,4226	6,4498	6,2458	72,7990
DTLS ECDSA CryptoAuthLib	25,5986	56,5797	0,4033	0,0869	0,0873	82,7558
DTLS ECDSA TinyDTLS	25,5986	56,5797	0,4226	6,4498	6,2458	95,2965

Table 6.5: Average execution time for phases and configurations.

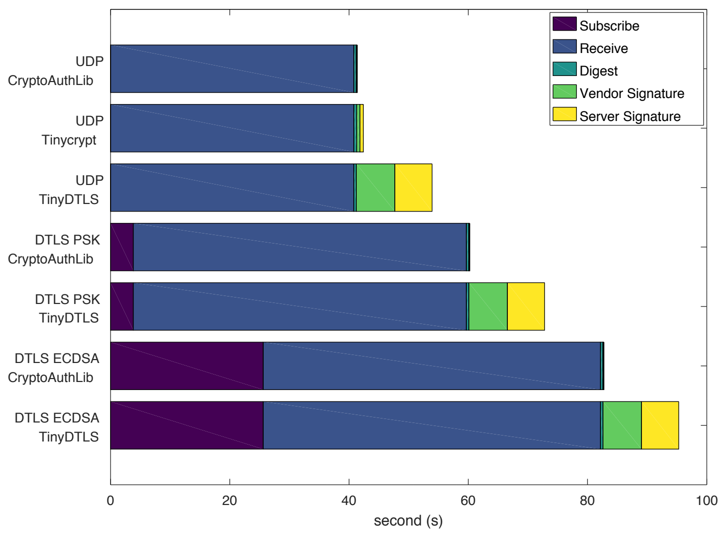


Figure 6.4: Average execution time for phases and configurations.

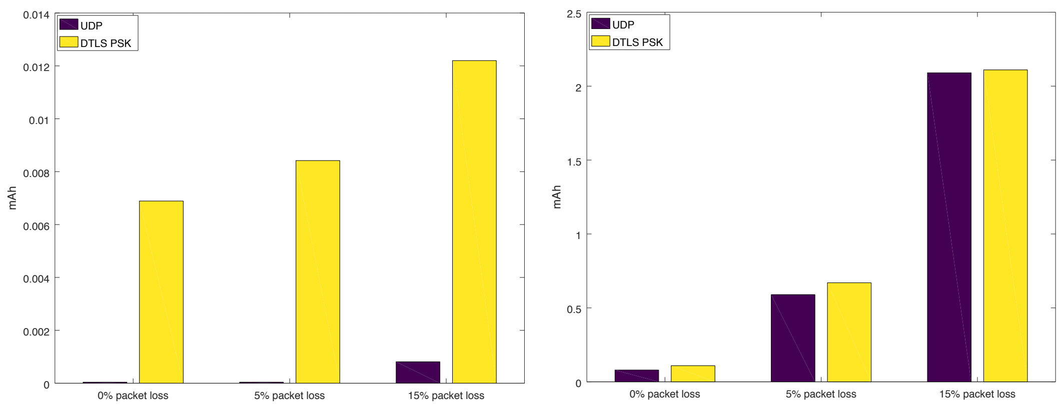
6.3 Energy Consumption

The evaluation of the energy consumption has been done since the solution, libpull, aims to be used on Class 1 devices battery powered, where the energy consumption is important to determine the lifetime of the device, since this kind of devices are tipically battery powered. We will evaluate the energy consumption in mAh, and to have a criterion for comparison, the capacity of the CR2032 (a typical cell coin used in many IoT applications) is of 240 mAh.

The measurement are performed using the Contiki-NG Energest [47] module, that provides a lightweight, software-based energy estimation, based on the number of clock cicles spend by the CPU and the Radio in each phase. It supports five different measurement types, three for the CPU (active, low power mode or deep low power mode) and two for the radio (transmit or receive). Dividing the value by the number of ticks elapsed during the whole measurement it is possible to obtain the percentage of time spent by the CPU or the Radio in each phase and estimate the power consumption based on the current required by the component.

Network and	Subscription (mAh)			Reception (mAh)		
packet loss	Min	Max	Avg	Min	Max	Avg
UDP (0%)	3,99E-05	4,05E-05	4,01E-05	7,86E-02	8,70E-02	7,97E-02
UDP (5%)	4,01E-05	4,63E-05	4,09E-05	5,28E-01	6,74E-01	5,90E-01
UDP (15%)	4,25E-05	7,71E-03	8,12E-04	1,96E+00	2,30E+00	2,09E+00
DTLS PSK (0%)	5,49E-03	7,97E-03	6,89E-03	1,09E-01	1,10E-01	1,09E-01
DTLS PSK (5%)	6,05E-03	2,23E-02	8,42E-03	5,12E-01	7,69E-01	6,70E-01
DTLS PSK (15%)	6,56E-03	4,18E-02	1,22E-02	1,94E+00	2,34E+00	2,11E+00

Table 6.6: Propagation phase energy consumption.



(a) Subscription phase. (b) Reception phase.

Figure 6.5: Propagation phase average energy consumption.

Verification Phase		TinyCrypt	TinyDTLS	CryptoAuthLib
Digest (mAh)	Min	1,12E-03	1,04E-03	9,90E-04
	Max	1,12E-03	1,04E-03	9,90E-04
	Avg	1,12E-03	1,04E-03	9,90E-04
Vendor Signature Verification (mAh)	Min	1,39E-03	1,58E-02	3,74E-04
	Max	1,39E-03	1,58E-02	3,75E-04
	Avg	1,39E-03	1,58E-02	3,74E-04
Server Signature Verification (mAh)	Min	1,39E-03	1,53E-02	3,74E-04
	Max	1,39E-03	1,53E-02	3,95E-04
	Avg	1,39E-03	1,53E-02	3,76E-04

Table 6.7: Verification phase energy consumption.

Similarly to the time evaluation, also the energy consumption evaluation of the propagation phase has been performed with three levels of packet loss to simulate a real environment. As shown in Table 6.7 and Figure 6.5, the energy consumption is highly affected with an high packet loss. In fact, with a packet loss of 15% the energy consumption is quite doubled during the reception phase (+96,18% in case of UDP and +94,84% in case of DTLS PSK). This, however, cannot be ascribed to libpull, since when encountered a network error, such as exceeded timeout, the receiver starts the reception requesting the image from the offset where the error occurred.

As already discussed in the time evaluation, also for the energy consumption the verification phase values are constant during all the evaluations, as shown in Table 6.7. When performing the verification with CryptoAuthLib the energy required by the ATECC508a has been added to the CPU and radio consumption. The component has also a small impact during the whole process due to its idle power supply current. When considering the energy of the verification compared to the total energy, Table 6.8 shows that the verification with TinyDTLS requires 25% more energy in case of no packet loss compared to tinycrypt and CryptoAuthLib. However, this value decreases to a negligible value of 1,33% in case of 15 % of packet loss.

Network and packet loss	TinyCrypt	TinyDTLS	CryptoAuthLib
UDP (0%)	8,37E-02	1,12E-01	8,15E-02
UDP (5%)	5,94E-01	6,22E-01	5,91E-01
UDP (15%)	2,09E+00	2,12E+00	2,09E+00
DTLS PSK (0%)		1,48E-01	1,17E-01
DTLS PSK (5%)		7,11E-01	6,80E-01
DTLS PSK (15%)		2,15E+00	2,12E+00

Table 6.8: Average energy consumption per configuration and packet loss.

Table 6.9 shows the energy consumption for each phase, summarizing it in the last column. This is also visually shown in Figure 6.6. Comparing the total energy required by the update process in optimal conditions with the capacity of a CR2032, the impact of 10 updates (value assumed as a possible number of updates during a device lifecycle) is very low and does not reach the 1% even when using the secure DTLS connection. However, the impact is much higher in case we consider the 15% of packet loss, requiring the 8,72% of the total energy when using UDP and tinycrypt and the 8,97% when using DTLS PSK and TinyCrypt. This shows the high impact of the network in the update process and a possible optimization would be to support delta updates and thus reduce the number of bytes to be transferred.

Network and cryptographic library	Subscribe (mAh)	Receive (mAh)	Digest (mAh)	Vendor Sig (mAh)	Server Sig. (mAh)	Tot. Energy (mAh)	Impact of 10 Updates
UDP CryptoAuthLib	4,01E-05	7,97E-02	9,90E-04	3,74E-04	3,95E-04	8,15E-02	0,34%
UDP Tinycrypt	4,01E-05	7,97E-02	1,12E-03	1,39E-03	1,39E-03	8,37E-02	0,35%
UDP TinyDTLS	4,01E-05	7,97E-02	1,04E-03	1,58E-02	1,53E-02	1,12E-01	0,47%
DTLS PSK CryptoAuthLib	6,89E-03	1,09E-01	9,90E-04	3,74E-04	3,95E-04	1,17E-01	0,49%
DTLS PSK TinyDTLS	6,89E-03	1,09E-01	1,04E-03	1,58E-02	1,53E-02	1,48E-01	0,62%

Table 6.9: Average energy consumption for all phases and configurations.

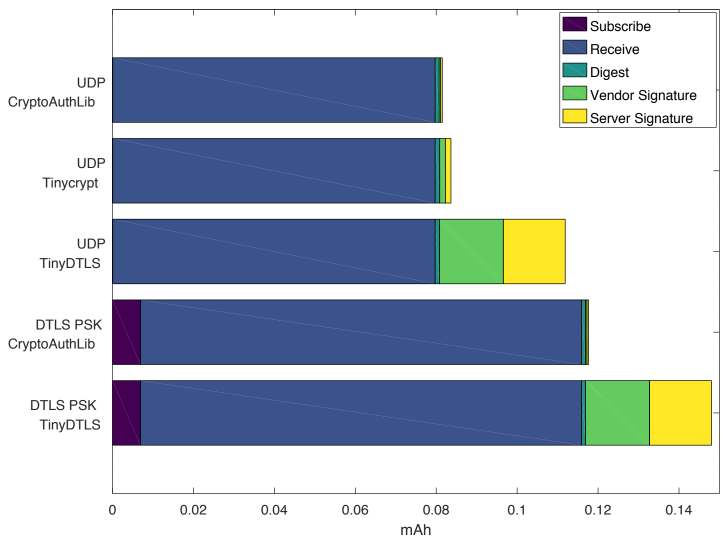


Figure 6.6: Average energy consumption for all phases and configurations.

Chapter 7

Conclusions

The IoT is a fast-growing technology that will change the way we interact with the objects of our lives in the next decades. The foundation principle is to make every object smart, interconnecting it with the Internet network to allow new kind of interactions with users and other machines. This is performed including a small computer inside of the object, able to empower it using sensors or actuators, but at the same time reducing the security, safety, and privacy of the involved users. Security represent a critical factor for this technology and must be included from the early stages of the development phase and also during the support phase, providing software updates.

In this thesis, we targeted the problem of software updates for constrained IoT devices. Starting from an analysis of the available update systems for IoT, we recognized the lack of an update system suitable for constrained devices, with a focus on a small memory footprint and low energy consumption. Moreover, we analyzed the update process identifying its components and security threats, and deriving the security requirements necessary to grant authenticity, integrity, and authentication during the update process. Considering the high number of devices and application requirements, we decided to build a library, highly modular and extensible, usable to create an update agent that better fits the needs of the platform and of the application. We reviewed all the requirements discussing the life cycle of a software update, moving through the designed architecture and all its components. We analyzed the design and implementation choices when building the library targeting two operating systems, Linux and Contiki. Finally, we evaluated the implementation in terms of memory footprint, execution time and energy consumption, analyzing all the possible combinations of network and cryptographic libraries supported.

As a conclusion, we think that libpull and its architecture is fully suitable for Class 1 devices, finding a tradeoff between security and portability that fits the device constraints, maintaining a high level of modularity that enables future extensions of the library to support other architectures and application needs.

Future Works

This study leaves some open research directions on how to optimize the size of the library and its energy efficiency. A possible approach to reduce the energy consumption is to support the delta updates, managing the reception of a patch that can be applied from one memory object to another. This feature is present in many update systems since it reduces significantly the size of the update, and also the load on the server. Other possible improvements consist in supporting different network protocols, such as MQTT for constrained IoT devices or HTTP for devices with more powerful resources.

Another improvement of the subscriber module, would be to integrate OMA LWM2M, that provides a standardized way to perform device management and already includes some features to manage the device updates. This would make the library suitable also in the context of a high number of devices, where the use of a management server is mandatory.

Bibliography

[1] *Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020*. URL: https://www.gartner.com/newsroom/id/2636073.

[2] *The search engine for the Internet of Things*. URL: https://www.shodan.io/.

[3] Nicky Woolf. *DDoS attack that disrupted internet was largest of its kind in history, experts say*. Oct. 2016. URL: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

[4] Eystein Stenberg. «Key considerations for software updates for embedded Linux and IoT». In: *Linux Journal* 276 (2017), p. 2.

[5] *OWASP Internet of Things Project*. URL: https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project.

[6] C. Bormann, M. Ersue, and A. Keranen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014.

[7] Hui Suo et al. «Security in the internet of things: a review». In: *Computer Science and Electronics Engineering (ICCSEE), 2012 international conference on*. Vol. 3. IEEE. 2012, pp. 648–651.

[8] Michael J Covington and Rush Carskadden. «Threat implications of the internet of things». In: *Cyber Conflict (CyCon), 2013 5th International Conference on*. IEEE. 2013, pp. 1–12.

[9] *SimpleLink Bluetooth low energy/Multi-standard SensorTag*. URL: http://www.ti.com/tool/CC2650STK.

[10] Aleksandar Milinkovi, Stevan Milinkovi, and Ljubomir Lazi. *Choosing the right RTOS for IoT platform*. 2015.

[11] Maria Rita Palattella et al. «Standardized protocol stack for the internet of (important) things». In: *IEEE communications surveys & tutorials* 15.3 (2013), pp. 1389–1406.

[12] G. Montenegro et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sept. 2007.

[13] T Winter et al. «RFC 6550: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks (2012)». In: *URL https://tools. ietf. org/html/rfc6550* ().

[14] C. Bormann and Z. Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016.

[15] Razi Hassan and Toheed Qamar. «Asymmetric-Key Cryptography for Contiki». MA thesis. Chalmers University of Technology University of Gothenburg Department of Computer Science and Engineering Göteborg, Sweden, July 2010.

[16] Oliver Kehret, Andreas Walz, and Axel Sikora. «Integration of hardware security modules into a deeply embedded TLS stack». In: *International Journal of Computing* 15.1 (2016), pp. 22–30.

[17] *Atmel CryptoAuthentication*. URL: https://www.microchip.com/design-centers/security-ics/cryptoauthentication/overview.

[18] Anthony Bellissimo, John Burgess, and Kevin Fu. «Secure Software Updates: Disappointments and New Challenges.» In: *HotSec*. 2006.

[19] Chuong Cong Vo and Torab Torabi. «A framework for over the air provider-initiated software deployment on mobile devices». In: *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE. 2008, pp. 633–638.

[20] *SHSH Blob: component of Apple digital signature*. URL: https://en.wikipedia.org/wiki/SHSH_blob.

[21] *Yocto Project*. URL: https://www.yoctoproject.org.

[22] *Mender.io: Over-the-air software updates for embedded Linux*. URL: https://mender.io.

[23] *SWUpdate: Software Update for Embedded Systems*. URL: https://github.com/sbabic/swupdate.

[24] Iulian Neamtiu et al. *Practical dynamic software updating for C*. Vol. 41. 6. ACM, 2006.

[25] *Fiat Chrysler recalls 1.4 million cars after Jeep hack*. July 2015. URL: http://www.bbc.com/news/technology-33650491.

[26] Goran Jurkovic and Vlado Sruk. «Remote firmware update for constrained embedded systems». In: *37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2014, pp. 1019–1023.

[27] Stephen Brown and Cormac J Sreenan. «A new model for updating software in wireless sensor networks». In: *IEEE Network* 20.6 (2006).

[28] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. *A remote code update mechanism for wireless sensor networks*. Tech. rep. 2003.

[29] Philip Levis and David Culler. «Maté: A tiny virtual machine for sensor networks». In: *ACM Sigplan Notices*. Vol. 37. 10. ACM. 2002, pp. 85–95.

[30] Ting Liu et al. «Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet». In: *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM. 2004, pp. 256–269.

[31] Jonathan W Hui and David Culler. «The dynamic behavior of a data dissemination protocol for network programming at scale». In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM. 2004, pp. 81–94.

[32] Miguel Salas. «A secure framework for OTA smart device ecosystems using ECC encryption and biometrics». In: *Advances in Security of Information and Communication Networks*. Springer, 2013, pp. 204–218.

[33] Shin-Ming Cheng et al. «Traffic-aware Patching for Cyber Security in Mobile IoT». In: *arXiv preprint arXiv:1703.05400* (2017).

[34] Eyal Ronen et al. «IoT goes nuclear: Creating a ZigBee chain reaction». In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 195–212.

[35] *Application Threat Modeling*. URL: https://www.owasp.org/index.php/Application_Threat_Modeling.

[36] *A/B (Seamless) System Updates*. URL: https://source.android.com/devices/tech/ota/ab/.

[37] Dan Goodin. *Update gone wrong leaves 500 smart locks inoperable*. Aug. 2017. URL: https://arstechnica.com/information-technology/2017/08/500-smart-locks-arent-so-smart-anymore-thanks-to-botched-update/.

[38] Koojana Kuladinithi et al. «Implementation of coap and its application in transport logistics». In: *Proc. IP+ SN, Chicago, IL, USA* (2011).

[39] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Oct. 2013.

[40] O. Bergmann. *TinyDTLS Software Library Implementation*. URL: https://projects.eclipse.org/projects/iot.tinydtls.

[41] *tinycrypt*. URL: https://01.org/tinycrypt.

[42] *Atmel CryptoAuthLib*. URL: https://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=CryptoAuthLib.

[43] *OTA for Contiki (CC2650 SoC)*. URL: http://marksolters.com/programming/2016/06/07/contiki-ota.html.

[44] *TOML: Tom's Obvious, Minimal Language*. URL: https://github.com/toml-lang/toml.

[45] *Travis CI: Continuous Integration tool*. URL: http://travis-ci.com.

[46] *Coveralls.io*. URL: http://coveralls.io.

[47] *Contiki-NG Documentation: Energest*. URL: https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-Energest.