UpdaThing: A secure and open firmware update system for Internet of Things devices

Tomás Alexandre Diniz de Pinho

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Miguel Filipe Leitão Pardal

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. Pedro Ricardo Morais Inácio

October 2016

Acknowledgments

The work developed in the scope of this dissertation, and its writing, could not have been possible without the help of numerous individuals who, by technical advisory or just plain motivation, were instrumental in its development.

I would like to thank Professor Miguel Filipe Leitão Pardal, whose orientation was crucial to the development of this work, for the long hours spent in orientation meetings, suggestions made and overall availability throughout the year. The Portuguese scientific community would benefit greatly from nurturing supervisors with such interest in their students and their work.

I am grateful to the Node.js, Linux and Buildroot open source communities for easing the amount of work that was put in the prototype, for producing readable documentation and reliable and usable code that powers millions of device worldwide.

I would like to thank my girlfriend Maria Inês Ferreira, whose motivational skills were extremely important in phases of stress and bewilderment and whose work ethic has taught me important life lessons. I would also like to thank my colleagues and friends at Instituto Superior Técnico, among other places, for allowing me to vent my frustrations and providing uplifting words at the extremest of hours.

I feel deeply indebted to my parents for not only providing me a place to live for all these years, while carefully paying my tuition, but for being enthusiastic about the work I do, despite its specificity, and for nurturing my engineering dreams since I was child.

Abstract

The interest of consumers and manufacturers in the Internet of Things (IoT) is increasing at a rapid rate, thus increasing device variety and sales. IoT deployments are usually made up of a set of sensors that collect information and connect to a central gateway device, which pre-processes and transmits data to a data center.

Software bugs are bound to appear in devices' software as well as in any codebase. Updating is usually done through replacing the whole device's memory with a downloaded image. A firmware update mechanism presents security challenges, as it can be used by malicious actors to compromise these devices in bulk.

In our work we propose, implement and evaluate an update management system for gateway devices that will provide manufacturers with an easy-to-deploy and easy-to-learn solution that will assure code authenticity (integrity and proof of origin), confidentiality and freshness in the update process, defeating the most relevant external security threats. Our system is open source, does automatic key and configuration management and includes a subsystem and configurations to generate Linux images, which differentiates it from similar systems in other works.

Keywords: Internet of Things, Firmware, Software update, Over-the- air update, Security, Public Key Infrastructure

Resumo

O interesse de consumidores e fabricantes pela Internet das Coisas encontra-se em crescimento acentuado, aumentando a variedade de dispositivos disponíveis e as suas vendas. Uma instalação de um sistema da Internet das Coisas inclui um conjunto de sensores que obtêm informação e a transmitem a um dispositivo *gateway* central que a pre-processa e a envia para um centro de dados.

A correção de erros de software, que aparecem em qualquer projeto informático, é normalmente conseguida através da substituição na totalidade da memória do dispositivo pelo conteúdo de uma imagem transferida. O mecanismo de atualização de *firmware* tem, portanto, um número de desafios de segurança, uma vez que pode ser usado por atores maliciosos para comprometer uma grande quantidade destes dispostivos.

Com o nosso trabalho, propomos, implementamos e avaliamos um sistema de gestão de atualizações para dispositivos *gateway* que oferece aos fabricantes uma solução fácil de instalar e aprender que assegura a autenticidade do código (integridade e prova de origem), confidencialidade e frescura no processo de atualização, anulando as ameaças externas de segurança mais relevantes. O nosso sistema é de uso livre, realiza gestão de chaves e configurações automaticamente e incluí um subsistema e configurações para geração de imagens Linux, o que o diferencia de sistemas semelhantes apresentados noutros trabalhos.

Palavras-Chave: Internet das Coisas, Firmware, Actualização de Software, Actualizações pelo ar, Segurança, Infraestrutura de Chaves Públicas

Contents

List of Tables

List of Figures

Acronyms

AC Alternating Current. 12

AES Advanced Encryption Standard. 6, 34

AOSP Android Open Source Project. 14

APS Application Support Sublayer. 7

ARP Address Resolution Protocol. 10

AWS Amazon Web Services. 38–40

CA Certificate Authority. 19–22, 31, 46

CAN Controller Area Network. 6

CDN Content Delivery Network. 30, 40

CPU Central Processing Unit. 12, 18, 46

DFD Data Flow Diagram. 31, 34

DH Diffie-Hellman. 19, 34

DNS Domain Name System. 10

DoS Denial of Service. 10, 18, 34

EBS Elastic Block Storage. 39

EPROM Erasable Programable Read-Only Memory. 11, 13

FAT File Allocation Table. 26

GPIO General-Purpose Input/Output. 2

GPU Graphics Processing Unit. 26, 27

HTTP Hypertext Transfer Protocol. 10, 13, 31, 36

HTTPS Hypertext Transfer Protocol over SSL. 10, 14, 19, 20, 22, 23, 32

IoT Internet of Things. 1–3, 5, 6, 8, 9, 11, 12, 17, 19, 30, 33, 37, 40, 43, 45, 46

IP Internet Protocol. 8

IPG Implantable Pulse Generator. 12, 13, 15

LAN Local Area Network. 9, 11, 13, 17, 19

MAC Message Authentication Code. 7, 12, 34

MITM Man-in-the-middle. 10, 11, 13, 18, 19, 34, 35

MRI Magnetic Ressonance Imaging. 6

MTU Maximum Transmission Unit. 37

NAT Network Address Translation. 31

OEM Original Equipment Manufacturer. 14

OS Operating System. 1, 2, 9, 10, 13–15, 18, 19

OTA Over-The-Air. 14

PAN Personal Area Network. 7, 8

PDA Personal Digital Assistant. 12, 13, 15

PID Process IDentifier. 25

PJL Printer Job Language. 8

PKI Public Key Infrastructure. 7, 11, 15

PPM Pulse-Position Modulation. 13

RAM Random Access Memory. 9, 18, 46

RCE Remote Code Execution. 34

ROM Read-Only Memory. 11, 26

SoC System on a Chip. 2, 3, 9, 13, 26, 43

SSD Solid State Drive. 39

TC Trust Center. 8

TCP Transmission Control Protocol. 36

TLS Transport Layer Security. 13, 19, 20, 28, 31, 33, 34, 36, 43, 45

UML Unified Modeling Language. 25

USB Universal Serial Bus. 2

WAN Wide Area Network. 13

WSUS Windows Server Update Services. 11

ZC Zigbee Coordinator. 7, 8

ZED Zigbee End-Device. 7, 8

ZR Zigbee Router. 7, 8

Chapter 1

Introduction

The Internet of Things term was coined by Kevin Ashton, executive director of the Auto-ID Centre, at MIT, in 1999 and denotes the connection of “Things” to the Internet, providing extended functionality [1]. An example of a “Thing” is an intelligent thermostat: it was previously disconnected from the Internet, but, by being connected, can now efficiently heat a home through the usage of real-time weather forecasts or by being user-activated. For the past few years, there has been a surge in popularity of the IoT term among consumers and manufacturers and it is now a well-known class of devices with multiple software libraries and development kits.

Architecturally, a typical Internet of Things (IoT) solution is composed of sensors, an IoT gateway (which may contain sensors in itself) and off-site storage/data processing services [2], as represented in Figure 1.1. Sensors are responsible for detecting events or measuring changes in the environment and passing on that information to the IoT gateway using a communication channel suitable for limited devices (e.g., ZigBee, Bluetooth, etc.). The IoT gateway, which runs an Operating System (OS) and specialized software developed and maintained by its manufacturer, does pre-processing of the collected data, by filtering and aggregating messages, and then sends the preprocessed data to the data center. The gateway also serves as a network protocol adapter, enabling sensors to communicate with devices that use Internet Protocols (IP) as their primary communication channels. The data center provides off-site storage/data processing services that may provide automatic data analysis and modeling whose results are generally consumed by web or mobile applications making them available to users.

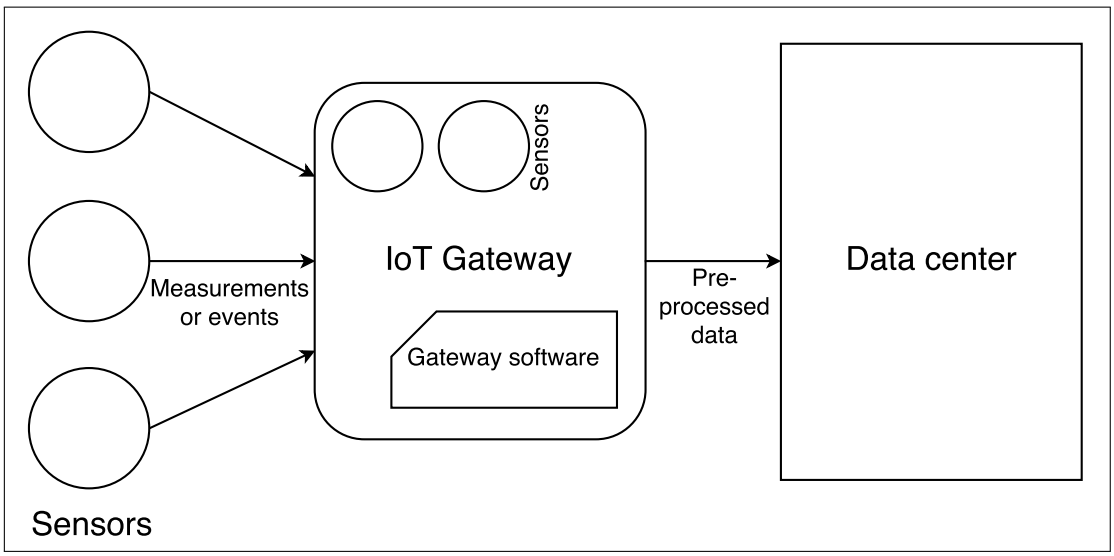

Figure 1.1: Architecture of modern IoT solutions.

As the IoT continues to grow, so does the number of newly-introduced connected devices that require manufacturer software. However it is an empirical finding of Software Engineering that all software has bugs and their number grows linearly with the amount of lines of code in a project [3]. Because of this, the development and deployment of software updates for the devices is necessary. These software updates are usually delivered to this class of devices via filesystem images that are written to the devices' main memory. However, most manufacturers are pressured to achieve a low time to market and end up jeopardizing their products' security by not carefully designing secure update mechanisms. Thus, a security challenge becomes evident: how to distribute these images in a way that protects them against tampering, and subsequent compromise of the device.

These software management concerns are shared by the IoT developer community. In regards to extra requirements for an agile IoT firmware update process, Dharmawan [4] states that providing centralized dashboards and processes to facilitate the monitoring and management of releases is a requirement for an agile IoT solution that accelerates secure software delivery. According to DZone, a well-known developer community, secure booting is an important property of security-aware IoT gateways [5] that, by checking the integrity of their firmware before booting, guarantee that its software has not been tampered with.

With this work, we propose, implement and evaluate a robust, secure and reliable firmware update system that easily integrates with common hardware architectures to increase the devices' security and to reduce manufacturers' costs. Since there are numerous and distinct chipsets and System on a Chip (SoC) offerings geared towards the development and integration of IoT gateway devices, we chose to develop an update management system targeted at IoT devices that run the Linux OS. Linux, by being open source, is more suitable for dealing with diversity, since device drivers are shared, more prevalent and easier to implement. These devices may be powered, for instance, by ARMv6 and ARMv7-based SoC packages. As all Raspberry Pi models belong to this category, that was one of the reasons why we chose them as a development platform. By being partly open source and relatively cheap (5-40$, depending on the model, as of the time of writing), the Raspberry Pi has been used by many companies as the SoC for their products. Media players[1], speech-controlled smart home interface devices[2], home sensors[3] and automatic cocktail mixers[4] are examples of products developed using Raspberry Pis.

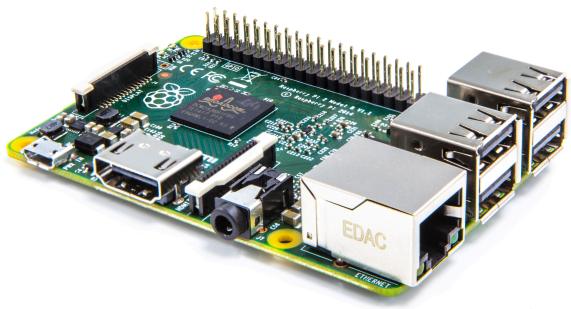

Figure 1.2: A Raspberry Pi model 2.

The Raspberry Pi may serve as an IoT gateway since it can connect to the Internet, via its built-in (dependent on the model) Ethernet port or Universal Serial Bus (USB) Wi-Fi adapter, and it can interface with other sensors, via other USB adapters, such as Bluetooth or ZigBee adapters. One may also connect sensors directly to the Raspberry Pi via its General-Purpose Input/Output (GPIO) pins, functioning as as

[1] http://fiveninjas.com/ - Five Ninjas, makers of the Slice media player
[2] https://www.athom.com/ - Homey, personal assistant for smart homes
[3] https://cubesensors.com/ - CUBESENSORS, home/office sensors to improve health, comfort and productivity
[4] http://partyrobotics.com/ - Party Robotics, makers of Robotic Cocktail Mixers

an IoT gateway with included sensors.

In summary, the cost of these devices, ease of use and popularity were the main reasons why we picked these SoCs for our work.

1.1 Main Goals

Our work explores the design and implementation of an update system for IoT gateways with focus on update source authentication, client authentication and firmware image confidentiality and integrity.

Our solution aims to satisfy the following requirements:

I Assure authentication of all parties participating in the update process: the updating client device and the update server.

II Assure secure communication between updating client device and update server, guaranteeing integrity and confidentiality of the transferred filesystem image.

III Provide an easy-to-use system for generating Linux filesystem images including the proprietary software developed by the manufacturer.

IV Perform automated key management to unburden the developers from doing it manually. This should include any keys and certificates used in the update process.

1.2 Document Overview

In this document we state the main goals of our work in Section 1.1 and go through software/firmware update systems that have been previously proposed and their contributions in Chapter 2. This chapter includes an analysis of the possible threats to a generic IoT firmware update system and a brief analysis of previously proposed systems specifically for the IoT. It also includes a summary of the related work and its key contributions. Our proposed solution is presented in Chapter 3. Chapter 4 discusses implementation details, and our final solution, and Chapter 5 presents our evaluation, including methodology, measurements and our analysis. Finally, Chapter 6 concludes this document.

1.3 Publications

The following publication describes part of the work done in this dissertation:

Tomás Pinho and Miguel Pardal, “UpdaThing: um sistema de atualizações de firmware seguro para a Internet das Coisas”, *Inforum*, 2016.

Chapter 2

Related Work

In this chapter, we present research work relevant to our research effort. Section 2.1 surveys the IoT and its security challenges found in the literature and Section 2.3 talks about recent attacks on embedded devices. Sections 2.4 and 2.5 go through related work on Software and Firmware updating, respectively, their definition, implementations and possible attacks.

2.1 The IoT and its challenges

Suresh et al. [1] conducted a review on the state of the art of IoT in their 2014 paper. It includes a broad overview of its history, predominant technologies and unsolved challenges. They showed that:

- This industry is experiencing rapid growth, as product citations in newspaper articles, television newscasts and specialty magazines have increased; IoT products are now in consumers' minds;
- The growth of the IoT poses some challenges:
    1. Compatibility issues, because of the myriad of different competing technologies that exist to solve the same problems, which, in conjunction with the lack of defined industry standards, causes a lack of interoperability between devices;
    2. Security issues, such as assuring the privacy of the user, integrity of communications, availability and authentication;
    3. Impacts on the environment, because proprietary devices are often unrepairable, get discarded once malfunctioning and are produced in quantities that create waste.

In regards to the security issues these devices present, which is the main focus of our work, we can infer that this type of devices often collect sensitive data on the user which must be kept private. Such data may be the position of the user, habits, environment (e.g., door openings) and network settings (e.g. Wi-Fi passwords). A few scenarios where IoT products are extremely useful, but their security challenges must be taken into careful consideration, were defined in the work of Suresh et al. [1]:

- **Industrial telemetrics and intelligent control**: by collecting and analyzing data in near real time, manufacturing plants are able to increase productivity, efficiency and safety. However, relying on insecure technologies may pose serious risks, when faced with external threat actors. Attackers may cause damage to operating machinery by wirelessly injecting control code or obtain privileged information by listening in on inter-machine communications.

- **Home automation and energy management**: by collecting and acting on data, home owners can automate multiple home appliances, such as light-bulbs, heaters, air conditioning systems and window-blinds. Although external attackers may cause limited damage, losing access to one's own home appliances may be exasperating. User data, however, is, in our opinion, the key aspect of security in this scenario. An external attacker may obtain sensitive information from the sensors in a home, which may reveal user habits. This sort of information may be used in user coercion.
- **Healthcare**: many medical devices are connected to the network and allow for convenient remote access by health professionals: infusion pumps, Magnetic Ressonance Imaging (MRI) machines and ultrasound machines, among many others. The danger of security vulnerabilities posed by this kind of devices is of the utmost importance, as a threat actor may be able to obtain privileged patient information such as health or insurance records, or, on a darker note, remotely end the life of a patient. This relationship between Healthcare and IoT security has been receiving attention from the media lately [6].
- **Vehicle telematics**: current generation automobiles contain networked components, connected using the Controller Area Network (CAN) bus that enables them to talk to each other. Engine control units, for instance, read values from a multitude of sensors within the engine bay and adjust the engine actuators accordingly. As all devices are connected to the same CAN bus and can typically talk to any other without authenticating themselves, a multitude of attacks are possible. Two security researchers managed to obtain remote access to a vehicle [7] and managed to achieve effective remote control by exploiting vulnerabilities in the entertainment system of the car, which is connected to the Internet by a mobile data connection, and then sending commands to other devices on the CAN bus. The security implications of such systems pose threats to passengers' well-being and lives.
- **Agricultural monitoring and control**: industrialized agriculture has adopted several IoT technologies leading to what is called *intelligent agriculture*. By leveraging sensors and data analysis, farmers can plant more accurately and successfully, while knowing how effective certain seed and types of fertilizer are in different sections of a farm. Sensors and actuators are, for instance, used in water dispensing and precision soil control (plant nutrients and *pH*). Attackers of these systems may be able to tamper with devices to cause crop withering, which results in financial loss for farmers.

These examples show how important and diverse certain IoT applications can be and the impact of possible attacks. Therefore, it is imperative to carefully study possible attacks to this class of devices and applications in order to be able to develop a secure firmware update system for it.

2.2 Security Mechanisms

Before heading to latter sections, we review a few basic security concepts:

Symmetric cryptography is a cryptographic system that uses the same cryptographic keys for both encryption of plaintext and decryption of its result (the ciphertext) [8]. The keys may be exactly the same or a simple known transformation must be performed to obtain the decryption key. These keys represent a shared-secret that must be disseminated by any two parties who wish to perform confidential communications. Common symmetric cryptographic algorithms include the Advanced Encryption Standard (AES) and Blowfish.

Asymmetric or Public-key cryptography is a cryptographic system that uses a pair of keys, one private and one public, for the encryption of messages [8]. When one of the keys in the key pair is used to

encrypt a message in an asymmetric cryptographic algorithm, the other one must be used to decrypt the ciphertext. The private key is known only to the owner and the public key may be disseminated to recipients of messages. This allows a recipient to mathematically authenticate the sender of a message when such a message, or its digest, is encrypted using the private key of the sender. Conversely, when one encrypts a message with the public key of another, the only key that can be used to decrypt the ciphertext is the private key of the other. Common asymmetric cryptographic algorithms include the Digital Signature Algorithm and RSA.

A hash is the result of a hashing function [8]. This class of functions maps data with an arbitrary length to a fixed-length digest and has a few properties: it is computationally difficult to find an input for any given function output (must be *one-way*) and to find two different inputs for which the outputs are equal (must have *strong collision resistance*). Common examples of hash functions are the SHA family of functions and MD5.

A Message Authentication Code (MAC) is a short piece of information used to authenticate a message, providing integrity and authenticity assurances on it, using a hash function that summarizes the entire message in a fixed-size hash and then running the same hash through an encryption algorithm [9] used to sign the message, using a pre-shared symmetric encryption key scheme or a public-key cryptography scheme.

A Digital Signature is a mathematical system that can demonstrate the authenticity of digital messages, documents or any other form of binary data. By coupling Public-key cryptography and a hash function, a message digest can be calculated, which assures the integrity of its contents with a high degree of certainty, and a signature of that message digest can be computed by using a Private key with a Public-key cryptography algorithm. Then, someone who wishes to verify the authenticity of a message and integrity can decrypt the signature with the Public-key of the sender, obtaining the plain-text message digest. By running the same hash function over the received message and comparing both message digests, the authenticity and integrity of a message is mathematically assured.

X.509 is a set of important standards for Public Key Infrastructure (PKI), it specifies formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm [10].

2.3 Recent attacks on embedded devices

As seen in Figure 1.1, modern IoT deployments comprise communications between sensors and gateways in order to relay collected information about the environment. Such communications may take place using radio protocols, such as ZigBee or BlueTooth. ZigBee [11] is an IEEE 802.15.4-based specification for a set of communication protocols used to create Personal Area Networks (PANs) with small, low-power digital radios. A ZigBee network is composed by three types of ZigBee devices: the Zigbee Coordinator (ZC) is tasked with network creation and acts as the central node in a star ZigBee network topology, the Zigbee Router (ZR) acts as an intermediate router, may implement application functionality and retransmits data from other devices, and the Zigbee End-Device (ZED) implements enough functionality to talk to a parent node (ZR or ZC) and consumes less power, being ideal for sensors.

ZigBee networks can be in any of three security levels: **Residential security** requires that a network key be shared among devices that encrypts traffic within a network, meaning that all devices that contain the key can decrypt packets sent within the network; **Standard security** adds a number of optional security enhancements over residential security, including an Application Support Sublayer (APS) link key that encrypts Application Data between source and destination devices, providing end-to-end encryption; **High security** adds entity authentication, and a number of other features not widely supported.

Being a widely used protocol in IoT devices, namely in the industrial IoT, it is a prime target for attacks. KillerBee, an IEEE 802.15.4/ZigBee security research toolkit, contains several tools that ease research of ZigBee-enabled networks and is used in both the following presented papers.

Vidgren et al. [12] described two attacks to ZigBee-enabled networks in their 2013 paper: an energy consumption attack and a network key sniffing attack. The first attack relies on abusing the reliance of a ZED on a parent device (router or coordinator) to remain awake and receive data packets. By impersonating the ZR or the ZC of the ZigBee-enabled system, the attacking device is able to reply, via broadcast or multicast, to all data polling requests of legitimate ZEDs with an affirmative answer and then not sending any data, keeping the ZEDs awake and polling every 100ms, draining their batteries. The second attack, a network key sniffing attack, is performed when ZigBee devices are using the Standard Security level. When the network key is not installed on legitimate devices on the ZigBee network, the Trust Center (TC), one of the devices in a ZigBee-enabled network who is responsible for the security management, sends the current network key unencrypted to the devices when they want to join the network, making it possible for an attacker to sniff the same network key and use it for eavesdropping and further attacks.

Olawumi et al. [13] described three attacks to ZigBee-enabled networks in their 2014 paper: a network Discovery and device identification attack, an interception of packets attack and a replay attack. The first attack exploits the ZigBee discovery process by sending ZigBee beacon request frames with the intent of obtaining configuration details of corresponding legitimate ZigBee devices, ZigBee Routers and Coordinators. ZRs and ZCs that receive the beacon request frame will respond by disclosing the PAN ID (which is an identification number that uniquely identifies a PAN and is common to all devices in the same network), Coordinator's or Router's source address, stack profile, stack version, and extended IEEE address information. The second attack, an interception of packets attack, is possible because most ZigBee networks do not use encryption or use a Standard security level. Therefore, packet interception is possible and feasible for an attacker who wants to eavesdrop on ZigBee network traffic and abuse the intercepted information for his/her own malicious purposes. Finally, the third attack, the replay attack, is carried out by intercepting network traffic and re-transmitting it, posing as the original sender. The authors note that the consequences of this kind of attack depend heavily on the content of the transmitted data, giving an example of a water spill gate at an hydro-plant: by replaying messages that instruct the gates to open by an x amount of degrees, the attack may drive the hydro-plant systems to a point of no return, possibly damaging equipment in the process.

Cui et al. [14] performed a firmware modification attack on HP LaserJet printers in 2013. They found out that the majority of printers from the LaserJet series implement a capability called HP-RFU (HP Remote Firmware Upgrade) that enables users to remotely update the firmware of a printer over the network by *printing* to it. The updating subsystem of a device recognizes the presence of a RFU binary by reading through the standard Printer Job Language (PJL) and reaching code in an undocumented language named *ACL* (the authors do not specify the meaning of the acronym), verifies the integrity of the binary and decompresses it. After reverse engineering the RFU binary structure, the authors were able to pack arbitrary executable code into a legitimate RFU package, emulate standard PJL and *print* the firmware to the internal memory of the printer, replacing the original firmware. Since the update procedures function as a normal print job and the printer is available on the network through most of its duration, the attack is stealthy and may go unnoticed. HP released a patch to fix this vulnerability, which introduces mandatory firmware update signature verification, but 2 months after its official release, the authors identified 90,847 unique public Internet Protocol (IP) addresses belonging to HP printers accessible over the IPv4 Internet and found that only 1.08% of those printers were patched against this vulnerability.

Domestic routers and access points are a way in to domestic IoT networks and may be used by at-

tackers to establish and maintain control over *IoT* devices. The guide to performing firmware modification attacks on home/office routers by Jones [15] summarizes a wealth of useful information for penetration testers. The guide covers everything from obtaining firmware updating access, via the web interface of the router, Telnet[1] and hardware access via serial interfaces, to firmware reverse engineering and modifications development. It is important to note that this is just an example of the trove of information that can be found accessible on the Internet about exploiting consumer-grade routers, which poses a huge security concern to hardware vendors and network administrators.

Baek et al. [16] describe the creation of a botnet consisting of Set-Top boxes by exploiting vulnerabilities found on these devices. The explored devices were composed of a SoC containing a 100 megabit Ethernet interface, volatile memory and a flash memory chip, where the OS image is kept and from which the device boots from. They found out that the OS image present in these devices contains a custom Linux 2.4 kernel, a custom BusyBox shell, a minimal web server for web-based configuration, a Telnet server for remote management, a set of update scripts and a copy of *wget*, which was determined to be used by the update scripts to download newer (updated) firmware images. The bootloader of these devices unpacks the root filesystem image stored in the flash memory chip to Random Access Memory (RAM) and proceeds to boot the Linux kernel from there. Although this bootloader has built-in integrity protection and checks the firmware image against a 1024-bit RSA key signed MD5 hash of the image, the authors discovered that the public key against which this verification is performed is also included within the OS image, which enables an attacker to bypass the integrity verification process by replacing it with another public key for which he/she has the corresponding private key. The authors were able to obtain root access via the open Telnet port of the device, by using a public default password, and were then able to install a number of known Unix command line tools to speedup the process of creating the botnet and to add tools to their arsenal for subsequent attacks. Finally, the authors, created a set of scripts that automated the creation of a botnet by exploiting the found Telnet access for available devices on the Local Area Network (LAN).

Given these examples, we can see that it is imperative that manufacturers consider the security issues that IoT their products create. However, even when using development methodologies that take security requirements into account, developers are human and make mistakes, making it virtually impossible to produce software, or firmware specifically, without bugs [3]. These bugs require fixes by the manufacturers that come to products in the form of software updates, whose mechanisms are object of study in the next section.

2.4 Software Updating

Software updating is the process through which a piece of software gets updated in order to be fixed or improved by the provider of the software. These fixes may include patching security vulnerabilities, other bugs and improving usability or performance.

Such a process is usually carried out by a Software Updater, a separate application or application component, that fixes the piece of software by replacing parts of the whole of the application with a newer version of the program. Software Updaters usually make use of two types of information: the update metadata and the update data itself. The metadata contains information about the update, such as its version, the software modules or sections to be replaced, provider identification, other possible constraints (hardware and software) and hashes of the update data (to ensure integrity when transmitting the data over a network). The update data can be a segment of code that replaces old code during the

[1] Telnet is a protocol that provides a bidirectional interactive text-oriented communications using a virtual terminal connection, being primarily used to permit remote access to a terminal on a given device

update process, or a complete binary (or a set of binaries) that replace older executables during the same process.

Secure firmware updating requires cryptographic techniques, such as Digital Signatures [17], to ensure the integrity of the deployed firmware images. Conventional software update systems found in OSs, applications and libraries have to implement similar or more complex procedures to provide the same guarantees.

Cappos et al. [18] analyzed possible attacks on the most widely used package managers for Linux Distributions and found out that most of them are vulnerable to a myriad of attacks. The different attacks include:

- **Arbitrary Package Installation Attack**: the attacker serves an arbitrary package to the client in place of a package that the user wants to install;
- **Replay Attack**: the attacker replays an old package version to an updating client, causing it to install a package with security vulnerabilities (when the package is being installed from scratch and not updated);
- **Freeze Attack**: similar to a Replay attack, but the attacker provides old package metadata to a client, causing its update process to effectively freeze at a point in time, making it prone, over time, to exploitation of security vulnerabilities present in the installed application(s);
- **Extraneous Dependencies Attack**: the attacker rewrites the package metadata to force the client to install extra dependencies that may contain security vulnerabilities;
- **Endless Data**: the attacker causes a Denial of Service (DoS) on the client by providing an endless stream of data that fills up the memory/disk of the client.

Cappos et al. [18] presented Binary Hashes, Digital Signatures and Timestamps as solutions to protect against these attacks. Binary Hashes and Digital Signatures effectively protect against Arbitrary Package Installation and Extraneous Dependencies by hashing the metadata of the package and binary contents and signing the hash, providing integrity and code authentication assurances. Binary hashes are usually computed by running a hash function through all of the bits that compose a file, mapping it to a fixed-length sequence of bits, a hash, that may be represented in a variety of ways (e.g. Base64 or Hexadecimal), and Digital Signatures are produced by running obtained hashes through a Public-key cryptographic algorithm with the private key of the signer, creating a signed hash, which may then be verified by any holder of the public key of the signer. Replay and Freeze attacks can be prevented by having timestamped metadata that is fetched from official and trusted repositories through a TLS-encrypted connection. The SSL/TLS protocol provides communications security over the Internet, allowing client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. By creating strong assurances around the authenticity of the transmitted metadata and by checking its timestamp, clients can protect themselves against this kind of attacks by refusing to install outdated packages, and, by disabling Man-in-the-middle (MITM) attacks, the clients make sure the metadata comes from a trusted source. Endless data attacks can be solved by limiting the metadata files' maximum download size and then dynamically limiting the download size of the package by checking its previously downloaded metadata.

Luettmann and Bender [19] analyzed and demonstrated how to exploit auto-updating applications that make use of Hypertext Transfer Protocol (HTTP). The attacks perform MITM carried out through Address Resolution Protocol (ARP) poisoning, Domain Name System (DNS) poisoning and packet injection techniques. In order to effectively defeat these attacks, one can enforce the usage of Hypertext

Transfer Protocol over SSL (HTTPS) which, by making use of SSL/TLS, assures update source authentication, confidentiality and data integrity, defeating any tampering attempts for as long as the private key of the server is kept secret. However, the updater application must properly validate the update server certificate, rejecting any other.

Even when a PKI is implemented and used in a secure software update system for code signatures, compromising the private key of the vendor usually results in a complete defeat of the system. To solve this problem, Samuel et al. [20] proposed a multiple-key signing system for package managers that is resilient to the compromise of a single key. The system defines a Root Key that delegates responsibility in three lower roles (using a signed *root.txt* file), that may re-delegate hierarchically: the Timestamp Key, the Release Key and the Targets Key. The first is responsible for signing a file (*timestamp.txt*) that specifies the latest release file (*release.txt*). Release files are signed by the Release Key and contain signed metadata hashes for all packages. Targets files (*targets.txt*) are signed by the Target Key and contain signed binary hashes for all packages. In this manner, if any of the three roles' keys is compromised, the Root Key can sign a new **root.txt** file that specifies a new trusted key, revoking the compromised key. If any of the roles' delegate keys (keys deeper in the chain) are compromised, the key of the role may sign a new version of the corresponding file, effectively revoking the compromised key.

So far, only open source software update systems have been discussed. Even though both Microsoft Windows and OS X are proprietary, some information regarding their update process is available online. Both employ code signing [21][2], but Microsoft Windows has been the target of detailed security research due to its major market share among enterprise servers and desktops [21]. Windows Server Update Services (WSUS) have been studied for possible attacks by Stone and Chapman [21]. They concluded that some WSUS deployments are vulnerable to MITM attacks, as Microsoft does not enforce SSL by default between updating machine and the WSUS server, leaving it as an optional configuration step for network administrators. Therefore, it is possible for an attacker on the corporate LAN to forge update metadata, thus creating a new update, and have the client machines run any binary signed by Microsoft as an installation command. Several such binaries are available and *PsExec*, a tool for running local/remote commands, is one of them, which allows for complete system takeover.

2.5 Firmware Updating

Firmware updating is a special case of software updating. It is the process through which an embedded device updates itself, by obtaining an image, or be provided with one, and flashing it to its main memory. Firmware, the sum of the operating system of the device, filesystem, and applications, is held in non-volatile memory devices such as Read-Only Memory (ROM), Erasable Programable Read-Only Memory (EPROM), or flash memory. It differs from software updating because, usually, the entire main memory of the device is replaced by a newer version, instead of updating individual files or applications by replacing them directly or their code segments. The process of firmware updating may, similarly to software updating, use two types of information, update metadata and the update data itself, whose purposes are similar to software updating mechanisms.

Konsek [2] defines three different approaches for the initial deployment of the firmware on gateway devices: *factory bootstrap*, *server-initiated bootstrap* and *client-initiated bootstrap*. *Factory bootstrap* means pre-installing the software on the memory of the IoT gateway during the manufacturing process. In *server-initiated bootstrap*, the software deployment server of the manufacturer communicates with the device and deploys the software to it. This approach scales better than *factory bootstrap*, but requires server-side initiation which implies that the server must be informed manually of the presence of the

[2]https://support.apple.com/en-us/HT204502

device. The third approach is *client-initiated bootstrap*. This method has the gateway as the responsible party for initiating its own software deployment by connecting to the central repository server and downloading and installing the software.

Secure firmware update mechanisms require cryptographic procedures and accompanying infrastructure. We consider **connected devices** the ones that have access to a regular network connection and can easily connect to the update server to update themselves. On the other hand, **disconnected devices** need user intervention or specific hardware to achieve a temporary network connection and/or conduct a firmware update.

2.5.1 Connected Devices

Nilsson and Larson [22], Itani et al. [23] and Costa et al. [24] developed systems for secure firmware updating in Intelligent Vehices, Constrained Network Devices and Digital Television Receivers, respectively. All implement different communication protocols, optimized for each usage scenario. These three systems implement code signing in different ways, the first and second via symmetrically signed MACs and the third via asymmetrically signed binaries using X.509 certificates.

Katzir and Schwartzman [25] acknowledge the usefulness of digital signatures to guarantee security in the update process of Smart Grid Devices, and propose a system that enables device signaling by utility companies. This signaling technique creates a device update window during which the device updates by receiving the firmware image on a specific frequency band. The signaling is done through the shifting of the Alternating Current (AC) frequency that can only be controlled by the utility company, making it extremely difficult for cyber threat actors to exploit the update process. This work demonstrates how very specific IoT deployment conditions can be turned into a security advantage.

Itani et al. [23] proposed an energy-saving method for firmware packet authentication: substituting individual packet MACs with a signed Bloom filter that contains all individual packets. A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set [26]. It works by feeding an element to k hash functions that return k array positions in a n-sized array. All those positions are set to *1* when adding the element. To test whether an element is in a set, feed it to the same hash functions and then test for the presence of all *1*s in the array. If any is set to *0*, the element is definitely not in the set. If no positions are set to *0*, there is a high probability the element belongs in the set. Using this process, the update process can preliminarily check for integrity with a lower, but high, confidence level and ask for retransmission of single packets, a technique which considers the restrictions of limited devices by reducing the number of required computations. When the process is complete, it then verifies the integrity of the entire packet chain, completely ensuring integrity.

2.5.2 Disconnected Devices

Apart from being disconnected from the network, requiring user intervention and/or special hardware to obtain a network connection in order to perform firmware updates, this category of devices is also more constrained in terms of Central Processing Unit (CPU) power or energy consumption, requiring the usage of lightweight cryptography or offloading of cryptographic processes to neighboring devices in order to perform secure firmware updates.

Cao et al. [27] developed a method for secure software updates in Implantable Medical Devices. Instead of removing the Implantable Pulse Generator (IPG) from inside the patient and performing the update manually, they propose a system that authenticates a Personal Digital Assistant (PDA) with the IPG via its controller, using a 128-byte code within a 5s timeout window (prevents brute force attacks), and

performs the software update via a lightweight communication protocol. As the connection between the PDA and the external controller of the IPG is made through a serial cable, and the connection between the controller and the IPG itself is done through low power consumption Pulse-Position Modulation (PPM), only physical or close proximity attacks are possible - given a known 128-byte authentication code.

2.5.3 Proprietary Firmware Update Systems

This section explores a few proprietary firmware update systems, their security mechanisms and possible attack vectors. We chose consumer-grade home routers and two other products, namely, the NEST Thermostat and the Google OnHub WiFi router, as they showcase different security maturity levels. Finally, we discuss Android, the leading OS for mobile devices.

Domestic routers have been investigated by security researchers throughout the years and their security in regards to firmware updates is still appalling [3] (as well as other mechanisms outside of the scope of this work). Initially, domestic home routers offered access to their firmware update procedures via their web interfaces, usually available on the LAN on port *80* via HTTP. These web interfaces accept firmware images (produced by the manufacturers and available on their websites) from the user via a web browser-initiated file upload and the routers' firmwares update themselves by flashing the new image to their permanent storage (EPROM, flash memory) and subsequently performing a reboot. As the routers' web interfaces may be exposed to the external Wide Area Network (WAN) and since most home router users do not change their default login credentials, these routers are wide open to firmware replacing attacks, allowing attackers to easily install malware on the devices[4].

In the last few years, home router manufacturers have developed new firmware update systems that allow the router to directly query the manufacturer' servers for new firmware updates[5][6]. Mobile and desktop applications have also been developed by the manufacturers for domestic router management, that may initiate firmware updates, to enable users to more conveniently manage their home networks [7]. At the time of writing, both Linksys and Netgear, two prominent home router manufacturers, were serving firmware update images from their websites using plain HTTP. One can assume such a connection is used by the routers themselves to perform Internet firmware updates, leaving the routers prone to MITM attacks that may be able perform firmware replacement.

The NEST Thermostat[8] and the Google OnHub WiFi[9], both products by Alphabet subsidiaries, employ advanced and secure firmware update systems. Both employ the usage of Transport Layer Security (TLS) for communication with the manufacturer and use it for downloading firmware update images. Both use digital signatures as means of performing integrity checks and update source authentication on the downloaded firmware update images and their boot-loaders only boot firmware images digitally signed by the manufacturers. Certificates' and their public keys' location have not been disclosed.

Android mobile phones usually contain an ARM-based SoC that runs the Android OS. The Android OS, which is developed by Google and based on the Linux kernel, operates on a Unix-like file-system structure on which several internal flash memory partitions (See Table 2.1) are mounted.

The boot-loader of the SoC, which is developed and maintained by the manufacturer of the device, is responsible for configuring the device to an initial known state and booting the Linux kernel[10]. By inserting integrity checking and authentication procedures in devices' boot-loaders (by using checksums

[3] http://www.securityevaluators.com/knowledge/case_studies/routers/soho_router_hacks.php
[4] http://krebsonsecurity.com/2015/01/lizard-stresser-runs-on-hacked-home-routers/
[5] http://kb.netgear.com/app/answers/detail/a_id/23442/~/installing-the-latest-firmware-on-a-netgear-router
[6] http://www.linksys.com/us/support-article?articleNum=140124
[7] http://www.linksys.com/us/support-article?articleNum=156285
[8] https://nest.com/security/
[9] https://support.google.com/onhub/answer/6309220?hl=en
[10] http://forum.xda-developers.com/wiki/Bootloader

Partition	Purpose
boot	Contains the Linux kernel and a minimal root filesystem (RAM disk) that contains the minimally required kernel modules to boot the rest of the system. The bootloader boots this partition and it is responsible for mounting the *system* and other partitions and for starting the runtime located on the *system* partition.
system	Contains system applications and libraries from the AOSP. During normal operation, this partition is mounted as read-only, and may only be changed by the *recovery* subsystem.
vendor	Contains system applications and libraries provided by the OEM. During normal operation, this partition is mounted as read-only, and may only be changed by the *recovery* subsystem.
recovery	Contains a second complete Linux system, including a kernel and the program that implements the update process, which is responsible for reading an update package and using its contents to update the other partitions.
userdata	Contains data saved by applications run by the user.
misc	Small partition used by recovery to store information about the ongoing update process in case the device is restarted while the OTA package is being applied.
cache	Temporary holding area used by a few applications (accessing it requires special application permissions) and for storage of downloaded OTA update packages. Other programs may use this space, taking into account that files can disappear at any time. OTA package installations may result in this partition being completely erased.

Table 2.1: Common Android partitions and their purpose.

or digital signatures), manufacturers are able to control what software is allowed to run on them, disallowing potential harmful versions of the firmware, such as tampered versions of the Android OS that give users more access to the device, including possible carrier unlocks. Although locked boot-loaders are in the interest of the manufacturers, some provide official mechanisms for boot-loader unlocking [11].

The Android OS official update procedures[12] consist in checking for updates and downloading firmware update images from the OTA update servers, which are written to the internal memory of the device. Then, its cryptographic signature is verified against certificates present on the device (bundled in `/system/etc/security/otacerts.zip`) and the user is prompted to install the update. If the user proceeds with the update, the device reboots into recovery mode, finds the update image via a commands file (`/cache/recovery/command`) and verifies the cryptographic signature of the package against the public keys present in the recovery partition (`/res/keys`). Upon successful verification, the Android recovery system updates all its partitions as necessary and then proceeds to reboot. HTTPS may be used to download the update, depending on the update metadata retrieved from the OTA update server, however, it would only provide in-transit confidentiality because code authentication is already performed, as the cryptographic signatures of the update package are throughly verified in a 2-stage process. The reader should note that this process, although officially recommended by the Google, may be re-implemented by the manufacturer of the device (who is responsible for adapting the OS to the device) or developer teams of other Android distributions (eg, CyanogenMod) and, although the recovery mode is usually used, the cryptographic signatures may not be verified and the download may not be done through an encrypted connection.

[11] http://developer.sonymobile.com/unlockbootloader/
[12] https://source.android.com/devices/tech/ota/index.html

2.5.4 Discussion

We will now discuss the problems that we found with the works studied in this chapter. Afterwards, we do a brief comparison of the studied systems and the security mechanisms they make use of.

Although the systems of Nilsson and Larson [22] and Cao et al. [27] implement procedures that guarantee in-transit confidentiality, as both use symmetric encryption with a shared-key between client and server, dumping the main memory of the devices easily gives access to the unencrypted firmware images. The system of Nilsson and Larson assumes presence of the shared-key in the vehicles on-board system, suggesting that strategies for distributing these keys should be proposed in future work.

Type of System	Update System	Authentication of Update source	Integrity (transit)	Confidentiality (transit)	Code signing	Type of encryption
Papers	Nilsson and Larson [22]	✓	✓	✓	✓	asymmetric/symmetric
	Costa et al. [24]	✓	✓	✗	✓	asymmetric
	Itani et al. [23]	✓	✓	✗	✗	symmetric
	Katzir and Schwartzman [25]	✓[13]	-	-	-	-
	CAO et al. [27]	✓	✓	✓	✗	symmetric / none
Proprietary	Home routers	✗	✗	✗	✗	✗
	Android OS	✓	✓	✗	✓	asymmetric
	NEST thermostat	✓	✓	✓	✓	asymmetric
	Google OnHub router	✓	✓	✓	✓	asymmetric

Table 2.2: Update Systems Comparison.

The system of CAO et al. [27] only guarantees Integrity and Confidentiality of the firmware image until the physician decrypts the AES-encrypted file using the PDA. Once it is decrypted, it is transmitted unencrypted to the IPG via its controller. Therefore, an attacker that can physically eavesdrop on communications between the PDA and the controller of the IPG or between the controller of the IPG and the IPG itself has access to the decrypted firmware image and can pursue tampering attacks. However, this situation is highly unlikely and will probably arise suspicion from health professionals as it requires extremely close proximity to the patient.

Finally, a brief comparison of the previously mentioned systems can be found in Table 2.2. As one can see, most systems rely on Code Signing (some via symmetric keys) to ensure firmware image Integrity and update source Authentication. Since we consider the act of Code Signing as signing a binary using a Digital Certificate (and related PKI), we chose not to take into account systems that perform techniques similar to code signing via symmetric keys. The systems verify the authenticity of an update by downloading a published hash of its contents, which is signed with the private key of the publisher, decrypting it with the public key of the publisher, hashing the obtained updated and verifying if both hashes are equal. Additionally, Confidentiality of the downloaded update package is commonly guaranteed through a symmetric cryptographic algorithm, along with the usage of image-encrypting symmetric keys. This

[13] Via the frequency-patterns-induced update window and possibly additional mechanisms (implementation dependent)

algorithm can be negotiated at connection time or just statically defined in the software that implements the update process.

Chapter 3

Proposed Solution

3.1 Data Flow

In order to propose a solution that correctly tackles the problem of secure firmware updates by implementing an update system for IoT gateways, we first chose to analyze a naive implementation which does not guarantee any advanced security properties and then list all its weaknesses. In Figure 3.1 one may find a Data Flow Diagram of a generic firmware update system for some IoT device, which models its data flows.

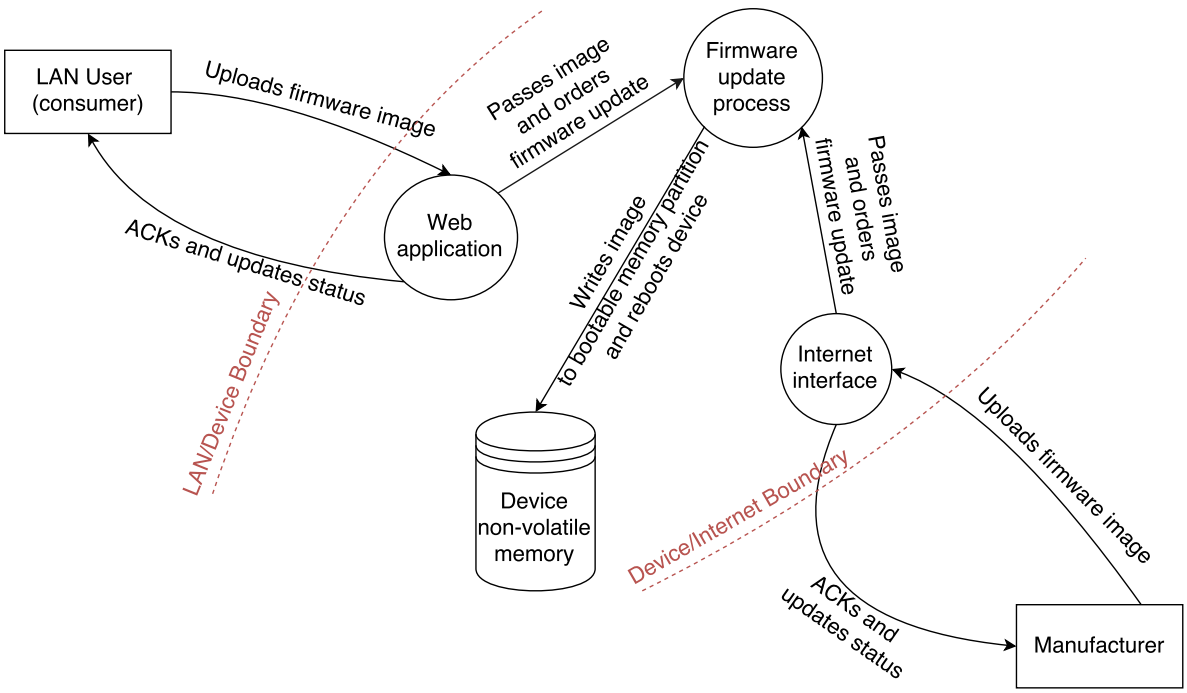

Figure 3.1: Data flow diagram for the firmware update process of a generic IoT device.

A firmware update process runs on the device every time it is called by a firmware update initiating process and is responsible for writing the image to the non-volatile bootable memory of the device and rebooting it. Two generic firmware update initiating processes usually exist: a component of the web application of the device that allows a LAN user to upload a firmware image and an Internet interface that allows the manufacturer to remotely initiate a firmware update by uploading a firmware image. We should note that, usually, the Web application is only available to the LAN and the Internet interface is

exposed to the entire Internet (which may or may not enforce authentication).

3.2 Threat Assessment

A threat modeling analysis, carried out using the **STRIDE**[1] framework, allows us to find vulnerabilities in a system and assess their risks.

Below is our analysis of Figure 3.1:

- **Spoofing**: considering that both the Web Application and Internet interface components usually may implement some form of simple authentication (username/password pair), it is often possible to obtain access to these devices by obtaining their default credentials. Even when the default username/password pairs are changed, most devices do not protect against brute-force attacks, allowing an external attacker to carry dictionary or brute-force attacks against the device. Manufacturers have to preselect username/password pairs to provide authentication features when a device is deployed and end users are usually tasked with changing those credentials. However, users generally do not change the default configuration of the device, leaving it unprotected to spoofing attackers. Consequently, the attacker can easily obtain access to the management interface of the device using these known credentials, allowing for escalation.

- **Tampering**: by performing MITM attacks on connections to the Web Application or Internet Interface, threat actors are able to intercept and tamper with in-transit firmware images or obtain user sessions that allow access to the management interface of the device. This enables them to tamper with the firmware and stored data of the device, leaving them in full control of the device.

- **Repudiation**: users may attempt to flash the device with non-working, or otherwise tampered firmware images, that cause the device to cease working properly. If the device has no way of logging actions of the user, he/she may repudiate its actions by claiming it to be the fault of the manufacturer and request otherwise deniable free costumer support or product replacement/refund.

- **Information Disclosure**: by performing MITM attacks on connections to the Web Application or Internet Interface, attackers are able to obtain in-transit firmware images, sensitive keys and other confidential data. Attackers may then be able to reverse engineer said firmware images and obtain any confidential piece of software, sensitive keys or business process that may be detrimental to the manufacturer. Device users may also be blackmailed by the attacker by threatening to divulge obtained confidential data.

- **Denial of Service**: as two interfaces are available to the network, the Web application and the Internet interface, it is possible for a threat actor to cause a device DoS by sending an uncopious amount of requests to the device, exhausting available OS sockets, device CPU or RAM. The device may also be flashed with a tampered firmware image that completely disables its intended behavior by a threat actor. Devices may be held for ransom by attackers as this sort of practice is getting more common with the current easiness of obtaining access to a botnet or purchasing DoS services.

- **Elevation of privileges**: by spoofing user identity or performing MITM attacks to tamper with in-transit firmware images, it is possible for an attacker to elevate his/her privileges by flashing a

[1] STRIDE is a system developed by Microsoft for organizing computer security threats, dividing them in 6 groups (whose initials give the system its name): Spoofing of user identity, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege

firmware image on the device that grants elevated privileges. For instance, by flashing a tampered Linux OS, which has altered */etc/passwd* and */etc/shadow* files, on a device, the attacker can easily obtain root access on it, establishing complete control.

Given this analysis, we assume our attacker capabilities are as follows: the threat actor may be present on both the LAN or Internet and is capable of communicating with the device, being therefore able to launch network-based attacks against it. We do not assume the attacker has unfettered physical access to the device and is therefore not able to tamper physically with the device by attaching any network monitoring equipment or able to read or write directly from or to the volatile or non-volatile memory of the device by attaching or soldering any related equipment, for instance.

3.3 Requirements Summary and Mechanisms

Given the requirements mentioned in Section 1.1, the manufacturers' intention to reduce costs and the time-to-market, and the security concerns that IoT devices create, our solution must accommodate all required tooling for a firmware deployment cycle, ranging from the production of the filesystem images, their signing, deployment and posterior device status checks. The solution must encompass means of assuring update source authentication, client authentication, in-transit image integrity and confidentiality. A firmware update system which does not implement mechanisms to provide these guarantees is vulnerable to attacks, such as device firmware tampering, firmware image disclosure, and reverse-engineering, both by obtaining original images and the first by propagating tampered images via MITM attacks.

Therefore, a solution that encompasses Public-Key Cryptography as the Code Signing method should be used in our proposed solution. This allows us to guarantee Integrity and Authentication of the downloaded firmware image and enables the update mechanism to verify the authenticity of the image by performing a validation of the trust chain of the certificate used to sign it. In order to ensure in-transit Confidentiality (as physically attacking the device and dumping its main memory easily reveals the decrypted firmware image in any case), session-generated symmetric keys should be used by leveraging TLS sockets - using the Diffie-Hellman (DH) Protocol as a symmetric key generation mechanism. This allows us to offload the burden of validating and negotiating cryptographically secure connections.

3.4 Key Management

Key management is one of the hardest, if not the hardest, problem to solve in any software update system. Our system intends to tackle this issue by including certificates (which include public keys) and Certificate Authorities (CAs) in the firmware images that get deployed to devices, effectively taking care of key management in a way that is transparent for the developers. We shall explain which responsibilities each component has in our architecture in the following section.

The image-signing key, which is used by the **Signing Server** for signing firmware images, and the **Update Server** certificate (used during communication via the HTTPS protocol) are included in the firmware images of the devices, which are flashed during manufacturing or during updating. This solves key management in regards to authentication, by the device, of the servers/services of the manufacturer.

We chose to also authenticate clients with the **Update Server**, while communicating via the HTTPS protocol. In order not to have a single client certificate for all the deployed devices, which would make it easier for an attacker to impersonate or use a device for nefarious purposes without being easily identified, we decided to have all the devices generate their own client certificates and signing them using a single CA which would be included and distributed by the **Signing Server** inside the firmware

images. This way, the server can authenticate clients by checking if their certificates were signed by the trusted device CA and use their key pairs to guarantee in-transit image confidentiality by negotiating a session key using the TLS protocol.

However, in the event of an attacker gaining physical access to the main memory of the device (which is beyond the scope of our protections, as discussed in the end of Chapter 2), we cannot guarantee device authentication in regards to identifying if the communicating device is legitimate. It could be some other device using a copy of a legitimate certificate or a certificate generated by a bad actor and signed by the same CA.

3.5 Architecture

Our proposed solution encompasses four components: an **Update Server** component, a **Signing Server** component, a **Client Update Daemon** component and a package of **Development Tools**. We chose to separate the **Update Server** component and the **Signing Server** component to allow for better network segmentation of services and better process isolation. This way, even if the **Update Server** is compromised, an attacker is not able to access the signing CA and is not able to sign his/her own firmware images.

The **Update Server** component is responsible for:

- Keeping track of available clients (see Figure 3.2) in order to know which devices are available for a firmware update;
- Notifying available clients of new updates (see Figure 3.3);
- Serving update images to asking clients (see Figure 3.4);

This server uses HTTPS as a messaging protocol, which guarantees client authentication, via the strict requirement of TLS client certificates, and transport security (in-transit confidentiality and integrity), by using cipher-suites negotiated in the TLS handshake. The device can also perform update source authentication by checking the certificate of the Update Server.

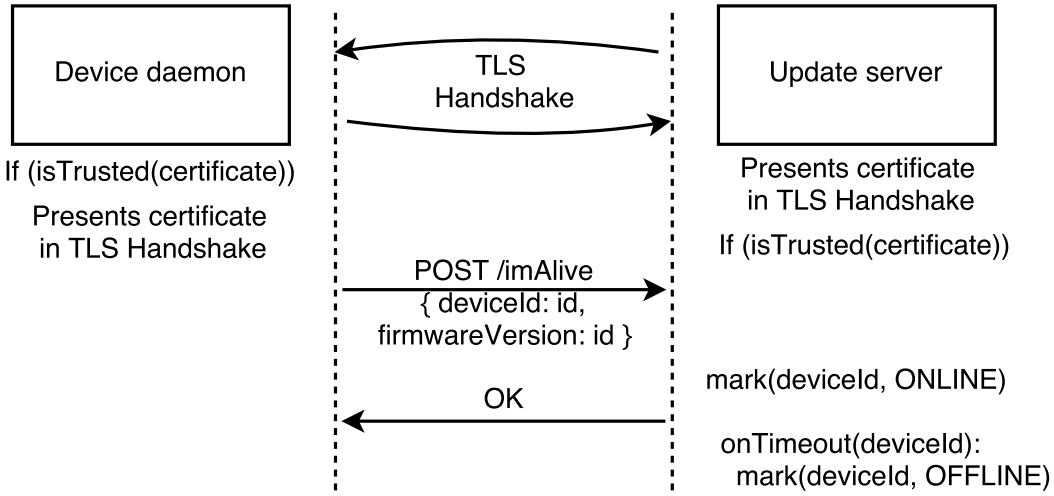

Figure 3.2: “I’m alive” messages and timeout procedure. The update server knows which devices are online to signal devices when a new update exists. This information is also stored in a database and can be used by the manufacturer.

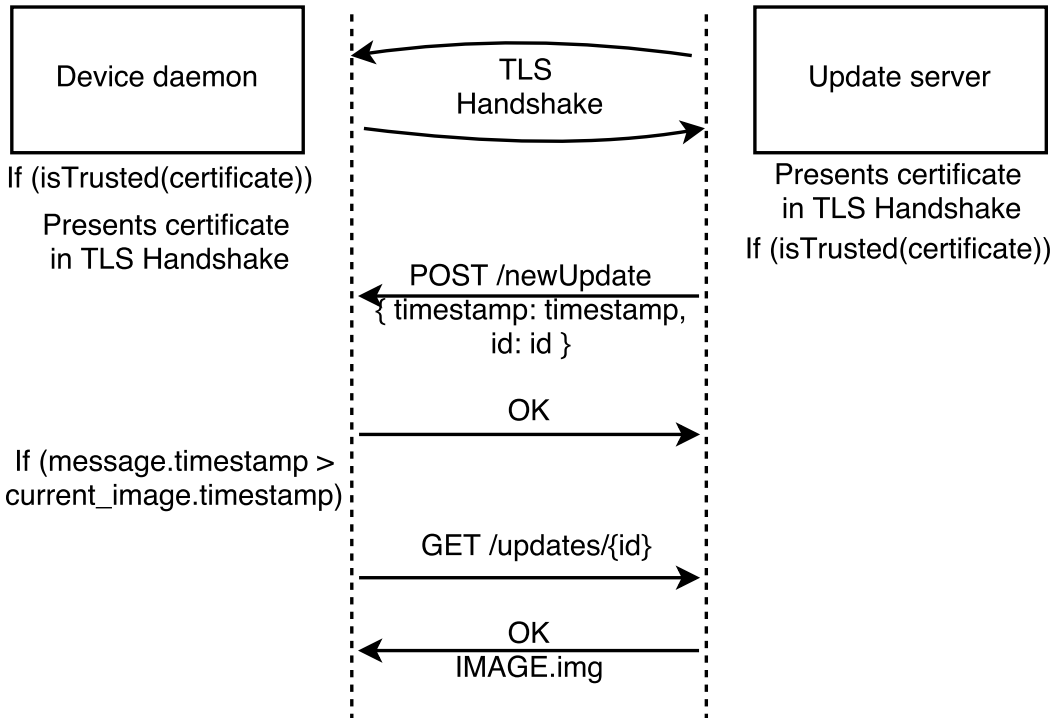

Figure 3.3: Update process initiated by the update server.

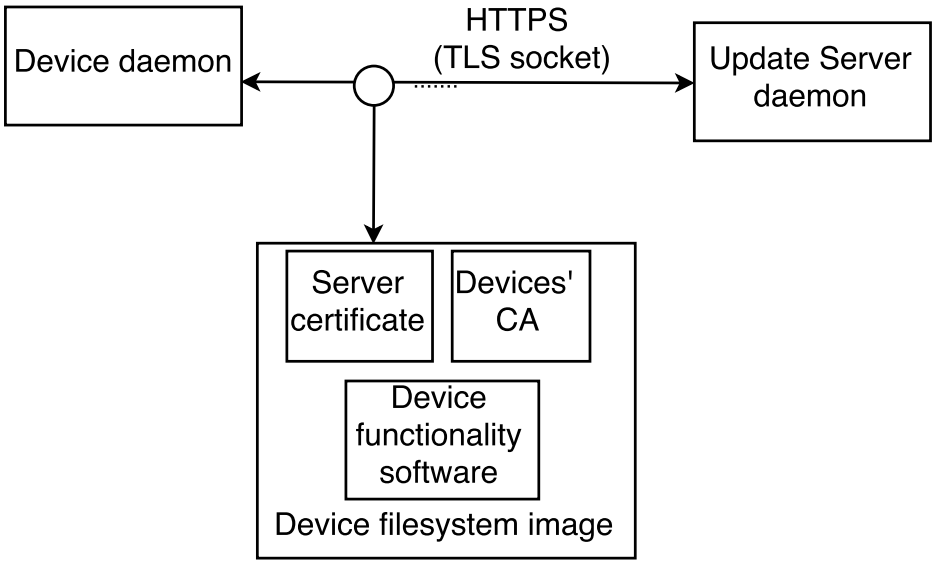

Figure 3.4: Usage of TLS for transmitting the filesystem image in the update process.

The **Signing Server** component is responsible for including the public key of the **Update Server** and a trusted device CA (its certificate and key pair) inside the filesystem images, which are uploaded to the server, using the provided tool, by the software developers of the manufacturer. The previously mentioned CA is trusted by the **Update Server** and is used by the client to sign its own public-key certificates. The **Signing Server** main responsibility is signing images with the Code Signing Key of the manufacturer and then transmitting signed images to the **Update Server** (see Figure 3.5). The separation of concerns between the **Update Server** implementation and the **Signing Server** implementation provides a service-oriented implementation that allows for network segmentation of involved machines.

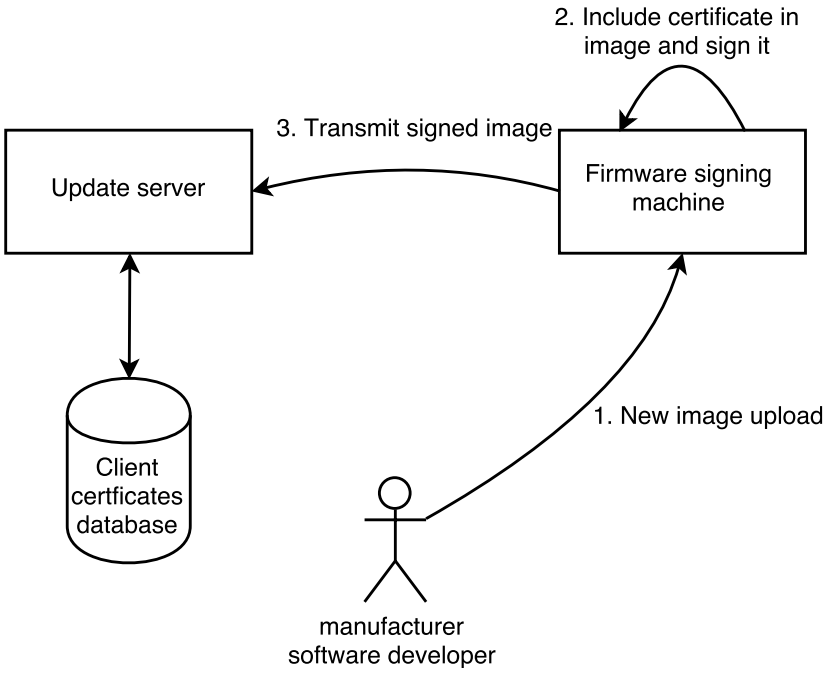

Figure 3.5: Signing process.

The **Client Update Daemon** is responsible for performing the firmware update by flashing received images to the main memory of the device, sending *I'm alive* messages to the **Update Server**, asking the **Update Server** for device updates on device boot (see Figure 3.6) and receiving update notifications from the **Update Server**. This daemon also implements HTTPS as a messaging protocol, guaranteeing update source authentication and transport security (in-transit confidentiality and integrity) and allowing for client authentication required by the **Update Server**. In order to allow for client authentication by the **Update Server**, the device generates its own certificate on its first boot, signs it with the included client CA (previously included by the **Signing Server**) and sends it to the **Update Server** for storage, which may be configured to ask for human validation before serving any update images to new devices. The **Update Server** may also ask the device to renew its own certificate, by generating a new one, signing it with the included CA and posting it to the server.

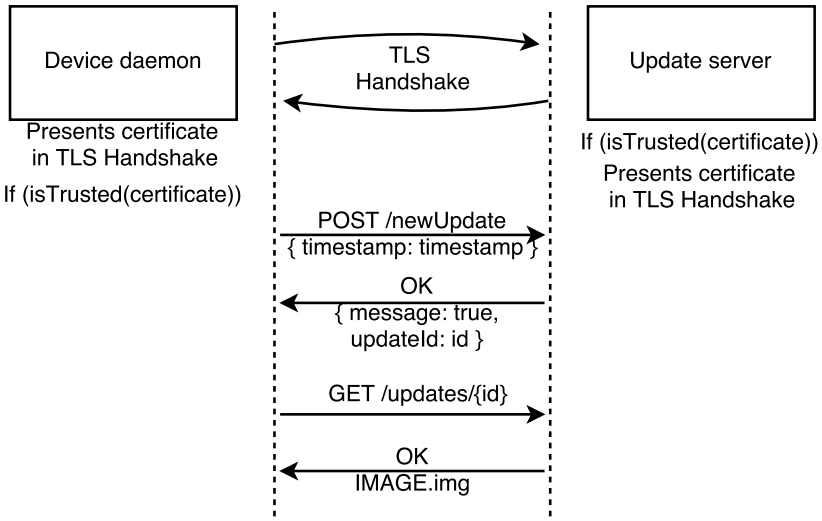

Figure 3.6: Update process initiated on device boot.

The **Development Tools** package contains programs and shell scripts that automate the creation of Linux filesystem images containing the set of software packages developed by the manufacturer and which implement the device functionality, and that automate the sending of the same images to the **Signing server** for signing and making them available to devices.

3.6 Summary

In this chapter, we specified the firmware update system that we aimed to implement. The solution we propose aims to satisfy the requirements we presented in Section 1.1. The first requirement, authentication of all parties participating in the update process, namely the updating client device and the update server, can be satisfied by the usage of HTTPS with server and client authentication enabled for all communications. HTTPS can also be used to satisfy the second requirement, assuring secure communication between updating client device and update server, guaranteeing integrity and confidentiality of the transferred filesystem image. The third and last requirement can be satisfied by the *Development Tools* package which takes care of the entire image generation and uploading lifecycle.

Guarantee	Authentication of Update source	Client Authentication	Integrity (transit)	Confidentiality (transit)	Code signing
Mechanism	TLS handshake	TLS handshake	TLS-negotiated Ciphersuites	TLS-negotiated Ciphersuites	Signing/ Verifying image
Credentials/ Parameters	Update server certificate	Client certificate	Negotiated cipher-suite parameters	Negotiated cipher-suite parameters	Signing Key

Table 3.1: Security requirements satisfaction: from guarantee to mechanism and required parameters.

Chapter 4

Implementation

UpdaThing is the name of this project and is an umbrella term for all of its components. UpdaThing encompasses a Linux distribution and its development tools, two distinct server implementations and a device daemon. Figure 4.1 presents a Unified Modeling Language (UML) deployment diagram of the system, including the Update Server, the Signing Server, the Device Daemon, and the Developer Tools.

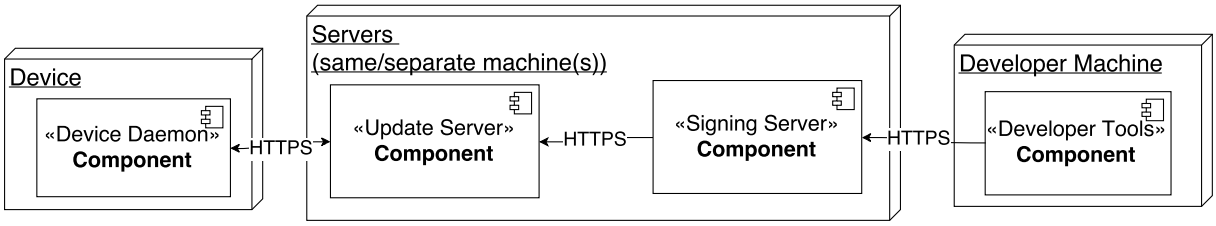

Figure 4.1: UpdaThing UML Deployment Diagram.

4.1 Linux Boot Process

Devices that run Linux have some sort of startup process where a bootloader is used to look for a Linux Kernel binary and load it into memory. After this step, the bootloader sets the program counter, or equivalent, register to the initial address of the Kernel, in memory, starting it.

After the Linux Kernel starts executing, it looks for the *init* system [28] in a predefined location (`/init` or `/sbin/init`, but is also be configurable as a Kernel parameter). This system is a special binary or script that is the first process started by the Kernel, being given Process IDentifier (PID) 1, and has the responsibility of starting the rest of the system: its services, user applications, etc. By delegating the initialization process to another program, the Kernel passes on responsibility for the rest of the initialization process to the programmer of the device. A set of conventional (and open-source) *init* systems are available, such as systemd, OpenRC and busybox-init. The latter is more suitable for embedded devices and we will be using it later.

The Linux Kernel, like the UNIX systems that came before it, exposes everything as a file in a tree-like structure [29]. This means that, for all usable given filesystems correctly configured to be used in a Linux system, there are their corresponding *mount points*, directories where their files are exposed. Linux itself has the concept of a *rootfs*, or root filesystem, a filesystem that is mounted at `/` and whose contents are directly accessible from the root of the file-tree. This rootfs must be mounted during the initialization of the system and must be accessible during it. However, it can be a cryptographically-protected volume,

a set of volumes behind a software RAID implementation or a network drive. This causes the famous "chicken or the egg" problem: how can we mount a special root file system without having its drivers, keys or general configuration parameters loaded at system start?

The *initramfs*, or init RAM file system, is a possible solution to this problem and was adopted by the Linux Kernel since version 2.5.46 [30]. It consists of a cpio ("copy in and out") archive of an initial root file system that is loaded to memory during the Linux startup process [31] [32]. This initramfs has initial drivers, configuration parameters and programs for performing various startup operations, such as loading more complex root filesystems. It contains an init system that is started, but may be explicitly replaced with a more complex init system present in the target rootfs.

4.2 Raspberry Pi Boot Process

The Raspberry Pi SoC has a Graphics Processing Unit (GPU) that is responsible for kickstarting the boot process [33], which we find rather interesting. When the Raspberry Pi is turned on, its ARM processor and SDRAM are off, but its GPU is on. It loads the first stage bootloader from a ROM embedded within the SoC. This bootloader reads the SD card, and loads the second stage bootloader (`bootcode.bin`) present in its first File Allocation Table (FAT) partition into the L2 cache, running it. The second stage bootloader then enables SDRAM and loads the third stage bootloader (`start.elf`) from the SD card into RAM and runs it.

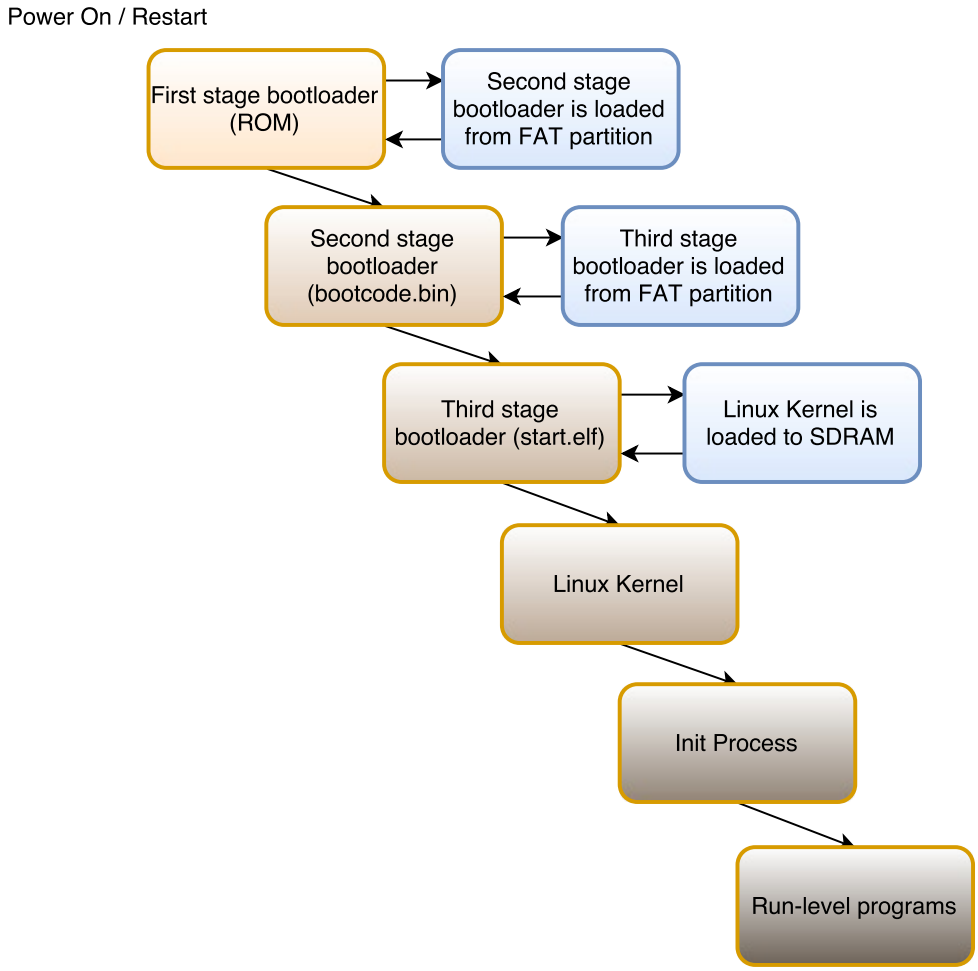

Figure 4.2: Summary of the entire Raspberry Pi boot process.

This last bootloader, `start.elf`, is the most important bootloader of the three. It is responsible for reading `config.txt`, a text file that has parameters for the GPU, such as Video modes, memory, console frame buffers, etc., and has parameters for the loading of the Linux Kernel, such as load addresses, device tree, console baud rates. Additional kernel parameters are read from the `cmdline.txt` file present in the same FAT partition and are passed to the kernel, these include the init script and rootfs locations, rootfs type, console devices, etc.

4.3 uLinux Distribution

We choose to custom build our own Linux distribution because we aimed to target devices that benefit from booting simple and small Linux environments and since we needed to control the Linux boot process of the device, its init system, to apply firmware updates. Buildroot is a set of Makefiles that allows for the configuration, compilation and generation of custom Linux filesystem images [34]. Buildroot allows us to generate minimal Linux filesystem structures that contain just the packages we need.

Our devices' permanent storage is split in three partitions: the first partition is where the minimal *initramfs*-backed kernel image is placed and booted by the Raspberry Pi, the second partition is where the main read-only root filesystem is flashed and the third one acts as a writable overlay filesystem over the contents of the second partition. The last two partitions are mounted on top of one another by our custom init process which we will describe later.

Our first fork of Buildroot allows us to generate a minimal *initramfs* that contains just the Linux kernel, a copy of Busybox (a small contained implementation of several Unix command line tools) and our custom init script. This custom init script takes care of the actual update image flashing, before the rest of the services are initiated, by checking a predefined filesystem location for an already verified update image, flashing it, deleting it and then proceeding to boot the rest of the system by switching the Linux root filesystem and starting the init system of Busybox.

```
ULINUX_DEVICE_DAEMON_VERSION = master
ULINUX_DEVICE_DAEMON_SITE = https://github.com/ulinux-embedded/ulinux-
    device-daemon.git
ULINUX_DEVICE_DAEMON_SITE_METHOD = git
ULINUX_DEVICE_DAEMON_LICENSE = MIT (core code)
ULINUX_DEVICE_DAEMON_LICENSE_FILES = LICENSE
ULINUX_DEVICE_DAEMON_DEPENDENCIES = nodejs
ULINUX_DEVICE_DAEMON_INSTALLDIR = $(TARGET_DIR)/opt/ulinux-device-daemon

define ULINUX_DEVICE_DAEMON_INSTALL_TARGET_CMDS
                mkdir -p $(ULINUX_DEVICE_DAEMON_INSTALLDIR)
                $(INSTALL) -D -m 0755 $(@D)/* $(
                    ULINUX_DEVICE_DAEMON_INSTALLDIR)
                # Install package node_modules
                (cd $(ULINUX_DEVICE_DAEMON_INSTALLDIR); $(NPM) install)
endef
$(eval $(generic-package))
$(eval $(host-generic-package))
```

Listing 4.1: BuildRoot Makefile example: ulinux-device-daemon package.

The second Buildroot fork is responsible for generating images of the root filesystem and contains directives to install packages such as network utilities, programming language interpreters and our own Device Daemon (see Listing 4.1). This is the Buildroot source code tree that is to be used by developers working on the devices' integration with manufacturer-developed applications. By writing a simple Makefile, any custom application may be added to the Buildroot build process and included in the generated image. The developer only has to checkout our repository, install compilation tools and type *make* on each file tree to generate working images.

4.4 Components

As previously described, our system is composed of four components: the uLinux distribution and its development tools, the Update Server, the Signing Server and the Device Daemon. In this section we will be fully describing their respective roles.

The uLinux distribution is used by the manufacturers' developers to generate working Linux file-system images. This greatly simplifies the process of getting software up and running on end devices, as it allows for an automated and repeatable build process that ultimately reduces the time taken in testing cycles. An included tool may be used to upload the resulting images to the Signing Server.

4.4.1 Signing Server

The Signing Server is our middleman between the developer, who produces firmware images, and the Update Server. It is responsible for receiving raw firmware update images from the machine of the developer, and including keys, certificates and configuration files in them. These artifacts are part of our automatic key management process and are used by the Client Daemon to authenticate images and the identity of the Update Server during TLS. To do so, the Signing Server spawns a Docker container that runs a custom shell script which in turn mounts the file-system image and copies the required files onto it (see Listings 4.2 and 4.3).

```
docker.run(config.image.container, [], process.stdout, null, {
    'Binds': [
      `${config.image.container_files}:/opt/files/:ro`,
      `${path_unsigned}:/opt/image.img`
    ],
    'Privileged': true
  }, function (err, data, container) {
      if (err) {
        logger.error('Got an error while running the keymgmt container',
            err);
        return;
      }
      signAndReply();
});
```

Listing 4.2: Spawning Docker container for the key management process.

```
#!/bin/bash
FILES_FOLDER=/opt/files
DEVICE_CA=$FILES_FOLDER/device_ca
DEVICE_CA_DEST=/opt/device_ca
SERVER_CACERT=$FILES_FOLDER/servers_ca.crt
SERVER_CACERT_DEST=/opt/ulinux-device-daemon/servers_ca.crt
SIGN_KEY_FILE=$FILES_FOLDER/signing_pubkey.key
SIGN_KEY_FILE_DEST=/opt/ulinux-device-daemon/signing_pubkey.key
IMAGE=/opt/image.img

mount ${IMAGE} /mnt/
cp -r ${DEVICE_CA} /mnt/${DEVICE_CA_DEST}
cp ${SERVER_CACERT} /mnt/${SERVER_CACERT_DEST}
cp ${SIGN_KEY_FILE} /mnt/${SIGN_KEY_FILE_DEST}
umount /mnt/
```

Listing 4.3: Script that runs inside the Docker container.

Docker[1] is an open-source project that allows a developer or system administrator to specify the deployment of Linux applications inside software containers, which is a method for server virtualization in which the kernel of an operating system manages multiple user-space instances, allowing for easier to manage and deploy server instances without resorting to virtual machines. We chose to perform the key management process inside a software container because we must mount file-system images and manipulate them. This would require more privileges for the user running the Signing Server instance, which could, ultimately, cause security concerns. This way, the process is isolated by the kernel and its execution and privileges can be more finely controlled.

After inserting the files which are part of the key management process, the Signing Server calculates a SHA512 hash of the resulting firmware image and signs it with its RSA Private key, including it in a Tar[2] package along with the image (see Listing 4.4). Finally, the Signing Server uploads this package to the Update Server.

```
let fileOnDisk = fs.readFileSync(path_unsigned);
const sign = crypto.createSign('RSA-SHA512');
sign.write(fileOnDisk); sign.end();

const privkey = fs.readFileSync(Path.join(__dirname, '..', config.image.
    signing_privkey), 'UTF-8');
const signedHash = sign.sign(privkey, 'base64');

let signedFile = fs.createWriteStream(path_signed);
let pack = tar.pack();
pack.entry({ name: 'signature.txt' }, signedHash);
pack.entry({ name: 'image.img' }, fileOnDisk);
pack.finalize(); pack.pipe(signedFile);
```

Listing 4.4: Signing and packaging firmware update image.

[1] https://www.docker.com/
[2] Tar stands for Tape ARchive and it is a file archiving format.

4.4.2 Update Server

The Update Server is the main point of contact for our IoT gateways with their updating infrastructure. It is responsible for notifying alive devices of new updates, by sending push notifications to the list of alive devices it maintains in the form of HTTP POST messages. It is capable of serving update packages by their ID, when requested, which is a standard HTTP GET followed by the download of the update package file. It receives I'm Alive messages from devices, which are stored in a MySQL database for logging purposes. These messages could further be extended for centralized logging collecting purposes, by just expanding the JSON object they contain. Finally, the Update Server is reponsible for responding to queries for newest update packages' IDs by the timestamp of the last installed image.

Special considerations must be had by the system administration team of the manufacturer to ensure this server is deployed in a scalable and distributed fashion in order to be able to handle the load generated by the current number of active devices. This may include the usage of a HTTPS-capable Content Delivery Network (CDN) to serve the firmware update packages and proper load balancing with layer-7 proxies such as HAProxy or nginx. In regards to the MySQL database usage, as the number of active devices grows, so does the on-disk size of the MySQL database. Proper strategies must be implemented in order to contain its growth, such as only keeping live data that refers to the last *n* days of activity and performing backups where required.

4.4.3 Device Daemon

The Device Daemon is tasked with periodically sending "I'm Alive" messages to the Update Server, being notified of new update packages when they are available, querying for newest update packages on the boot of the device and downloading and verifying update images. The update images are verified by opening the Tar archive, hashing the included firmware update image using SHA512 and verifying the included signed SHA512 hash by using the Signing Public-Key of the Signing Server that is installed on the permanent memory of the device (see Listing 4.5).

```
// 'signature' and 'image' variables are set while processing the tar
   package stream
const verify = Crypto.createVerify('RSA-SHA512');
extract.on('finish', () => {
  if (!signature || !image) {
    reject(new Error('Signature␣or␣image␣is␣missing␣from␣downloaded␣tar
       ␣file.'));
  } else {
    verify.write(image);
    verify.end();
    if(verify.verify(signing_key, signature, 'base64')){
      resolve(image);
    } else {
      reject(new Error('Image␣signature␣was␣not␣successfully␣verified.'
         ));
    }
  }
});
```

Listing 4.5: Authentication of the downloaded firmware image.

Afterwards, the Device Daemon reboots the Device, passing on control to the init script described in Section 4.3, which is responsible for the actual flashing of the image to the boot partition of the device. The Device Daemon also attempts to establish port forward mappings with the router it is connecting to through the usage of UPnP [35]. Universal Plug and Play (UPnP) is a suite of network protocols that allow devices connected to the same network to discover each other without human intervention and to automatically establish network services. This enables the Device Daemon to receive push notifications from the Update Server directly, even when it is behind a compatible residential router that has a firewall and implements Network Address Translation (NAT).

In order for authenticated communications to take place between the Device and the Update Server, the Device Daemon has access to a CA common to all devices with which it is able to sign client or server certificates for itself and which is included in the image by the Signing Server. When the Device Daemon starts, it checks for the presence of the required certificates and their respective Public-Private key pairs and, if any of them are not found, it proceeds with their generation and posterior signing using the device CA. Therefore, the Update Server only trusts certificates signed by this CA, allowing access to its endpoints only to devices with valid certificates.

4.4.4 Prototype Implementation

The Signing Server, the Update Server and the Device Daemon were implemented in Node.js. We chose this platform because JavaScript is a quick prototyping language and Node.js provides useful APIs out-of-the-box, such as implementations for HTTP, TLS and a Cryptography library (based on OpenSSL bindings) that more than fits our use case. Moreover, the vibrant open-source community of Node.js has developed libraries implementing functionality we required (i.e., integration with the Docker API) in a way that is not convoluted and could easily be integrated in our project.

4.5 Protocols

In this section we would like to delve deeper into the endpoints exposed by each of our components, the messages that are sent and received and the sequential logic behind them. As explained before, these endpoints are exposed via HTTPS and benefit from all the guarantees provided by TLS. An additional Data Flow Diagram (DFD) may be consulted in Figure 5.1. The implemented endpoints are as follows:

- Update Server
  - GET /updates/{id} - Download an update package by its ID. Must provide a valid device certificate.
  - POST /newUpdate - Get the newest update package after given timestamp. Post body: { timestamp: UNIX Timestamp (Number) }. Must provide a valid device certificate.
  - POST /updates/ - Upload a new update package file. Must provide a valid server certificate and Signing Server Auth token (in HTTP header).
  - POST /imAlive - Send I'm Alive message. Post body: { deviceId: UUID (String), firmwareVersion: Number, port: Number }. Must provide a valid device certificate.
- Signing Server
  - POST /updates/ - Upload a new update image file. Must provide a valid developer Auth token.

- Device Daemon
  - POST /newUpdate - Notify of a new downloadable update package. Post body: { id: Update ID (Number), timestamp: UNIX Timestamp (Number)}

Example scenarios for sequential usage of these endpoints are as follows:

- Device Daemon attempts to update on boot: the Device Daemon calls the endpoint of the Update Server “POST /newUpdate”, if an ID was returned, it then calls “GET /updates/{id}” to download the update and proceed with the update process.
- Device Daemon is notified of a new update: the Update Server calls the endpoint exposed by the Device Daemon “POST /newUpdate” and then the Device Daemon calls “GET /updates/{id}”, downloading the image and proceeding with the update process, if the received timestamp is higher than the one for the currently installed firmware version.

4.6 Summary

In this chapter, we discussed our implementation of the firmware update system that we specified in Chapter 3. The solution we have implemented has a modular approach and is comprised of two services, a Device Daemon and a set of Development Tools, targeting the Raspberry Pi 2 development platform.

HTTPS is used as a communications protocol between all system entities and assures integrity, authentication and confidentiality. Authentication of all entities is being assured by the usage of trusted X.509 certificates and Certification Authorities. Transmitted update images are further authenticated by the usage of Digital Signatures via the application of the SHA512 hash function and the RSA Public-key cryptography algorithm for the signing phase. The digital signature of an image is transmitted along with it to allow for this verification.

The process of creating, uploading, signing and providing Linux firmware images for the Raspberry Pi is fully automated by our system modules, which take care of key management and signing automatically, since the Signing Server is capable of mounting these images and placing all security payloads in them, signing it and packaging it for the Update Server, which then makes them available to updating devices.

Chapter 5

Evaluation

5.1 Methodology

Our solution was evaluated in several distinct ways: if and how the initial requirements were satisfied, formal threat analysis, to evaluate how well our solution fares against external threats, network overhead and energy consumption measurements that will be helpful to understand the trade-offs necessary for the security guarantees that our system provides, a deployment cost estimation analysis that attempts to calculate a cost for a lifespan of a device using this system, and usability surveys that established how straightforward and useful our system is to IoT product manufacturers and developers.

The threat analysis phase was comprised of a **STRIDE** analysis that formally analyzed how our system fares against external threat actors, by identifying all possible weaknesses and associating them with the mitigations we have adopted for each one of them. We therefore compared our solution to a baseline implementation where no security is enforced by producing tests that demonstrate the prevention of possible attacks.

Network overhead measurements were performed on all network-related actions the system may take: downloading a update image, sending *I'm Alive* messages, requesting for new update metadata, etc. Energy Consumption measurements were performed on all states the device may be in: booting, requesting for new update image metadata on boot, downloading a firmware update image, verifying and flashing it, etc.

A cloud deployment cost analysis attempts to compute an estimate of the cost of a device from the bottom up to determine its lifetime and per update cost, when using a popular cloud provider. By taking into account the previous network overhead measurements, new data on how much stored data costs, and the pricing for the cloud provider, we computed costs per messages sent, data stored and updates downloaded to be able to sum up the entire cost of each device for a period of 2 years, a typical warranty lifetime of a product.

The usability surveys phase attempted to establish how our solution fares in the manufacturers' and developers' minds by analyzing how easy to use and deploy the developed system really is.

5.2 Requirements Satisfaction

In this section we give a brief summary of which solution requirements were satisfied and how, which are stated in Section 1.1.

Requirement I is satisfied by the usage of X.509 certificates signed by trusted Certificate Authorities in the authentication part of the TLS protocol on which HTTPS operates. TLS guarantees authentication

of all parties participating in the process since all peers exchange X.509 certificates which contain RSA Public keys. The property of a given RSA Private Key is verified during the TLS handshake, since connection parameters must be signed using it (e.g. Ephemeral DH keys). Clients and servers must provide certificates for authentication, which disables any MITM attack.

Requirement II is satisfied by the use of TLS which guarantees confidentiality and integrity of transferred data. TLS negotiates a cipher suite in its handshake phase and allows both client and server to agree upon it as well as its parameters. Since our servers and clients are running up-to-date versions of OpenSSL bindings in their Node.js runtime environments, the agreed cipher suite is by default AES, with a key size of 128 bits, and the agreed MAC algorithm is SHA256, which is signed by the MAC key of the connection derived in the handshake phase.

Requirement III is satisfied by the Development Tools that were developed as part of this system that allow the developers to automatically configure, generate and upload complete Linux filesystem images using Buildroot. We repackaged Buildroot, produced our own configuration files for it and wrote a few automation scripts, enabling the manufacturer to build complete working filesystem images for the Raspberry Pi out-of-the-box. By writing a simple Makefile, developers are able to easily package any application in the produced Linux filesystem images which greatly simplifies the packaging process. An included tool allows the developers to easily upload the images to the Signing Server to undergo the automatic key management process before being signed and uploaded to the Update Server.

Finally, Requirement 4 was satisfied by the process of including all required keys, certificates and configuration files before signing a given update image in the Signing Server. The Signing Server receives the uploaded image, spawns a Docker container that runs a custom Bash script which in turn mounts the update image and copies all required configuration files, certificates and keys to the required location in the filesystem of the image. The Docker container stops after finishing execution of the script and the resulting image is then signed by the Signing Server.

Requirement	Satisfied
I - Assure authentication of all parties	Yes
II - Assure confidentiality and integrity	Yes
III - Easy to way to produce Linux images	Yes
IV - Automatic Key Management	Yes

Table 5.1: Summary of requirements satisfaction.

5.3 STRIDE

The **STRIDE** Threat Model is an industry standard for evaluating systems in regards to their security. **STRIDE** stands for Spoofing, Tampering, Repudiation, Information disclosure, Denial of dervice and Elevation of Privilege. In this section we perform a security analysis of our system and its endpoints.

In Figure 5.1 we present a DFD for all components of UpdaThing and their respective endpoints. Considering a scenario where some or all of the endpoints are not protected by X.509 certificates as an authentication mechanism or the TLS protocol as a mean to guarantee integrity and confidentiality in the update process, we can think of the attack situations presented below:

- GET /updates/{id} - From Device to Update Server - Device controlled by an attacker could download update images. Malicious actor could perform a MITM attack and force a device to download a given update image (could cause a DoS, achieve Remote Code Execution (RCE) if the key of

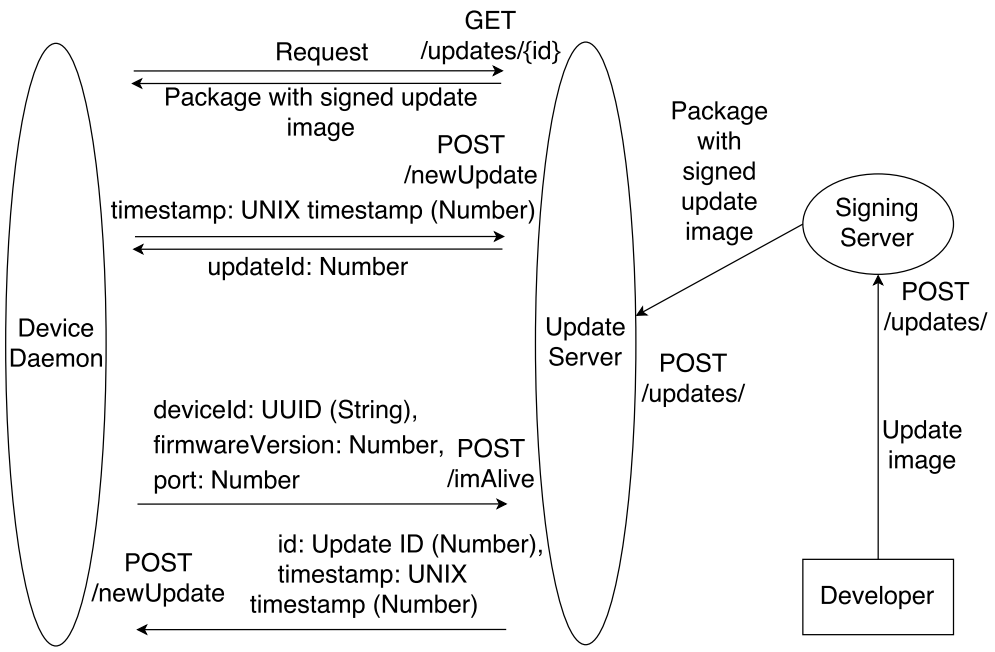

Figure 5.1: Data Flow Diagram of components of UpdaThing.

the signing server has been compromised and images signed with the correct keys were produced or even force a rollback to a previous legitimate vulnerable firmware version). **Spoofing** a device and update server identity, **Tampering** with a update image of a client, **Information disclosure** of firmware update image and **Elevation of privilege** in regards to access to the updating device are, thus, possible.

- POST /newUpdate - From Device to Update Server - Threat actor could perform a MITM attack and force devices to download update images by their IDs (forcing rollback to a previous legitimate vulnerable version). **Spoofing** the identity of the Update Server and **Tampering** with the update ID are possible.

- POST /updates/ - From Signing Server to Update Server - Attacker could upload malformed or badly signed images as well as sniff and obtain legitimate firmware images. **Spoofing** Update Server or Signing Server identities, **Information disclosure** of firmware update images and **Denial of service** by uploading malformed/badly signed images are, thus, possible.

- POST /imAlive - From Device to Update Server - Malicious actor could perform a **Denial of service** to the Update Server by injecting a high number of malformed I'm Alive messages and **Spoofing** devices' identities, because the Update Server notifies available clients when a new firmware update is available.

- POST /updates - From Developer to Signing Server - Threat actor could upload malicious images to get them signed and installed on devices, or sniff unsigned legitimate firmware images. **Spoofing** identities of a developer or the Signing Server, **Tampering** with installable firmware update images, **Repudiation** of uploaded images and **Information Disclosure** of firmware update images are possible.

- POST /newUpdate - From Update Server to Device - Attacker could pose as the Update Server and notify the device of an outdated update image, forcing it to rollback its software version. **Spoofing** the identity of the Update Server and **Tampering** with installed firmware version are possible.

Taking into account the thorough **STRIDE** analysis we have performed, we believe that the usage of TLS as part of HTTPS defeats all mentioned attacks as long as the certificates, their keys, the Signing Key and the developer Auth token are not obtained by an attacker. We conclude that our is system safe and protected against all enumerated attacks.

5.4 Network Overhead

We measured the network overhead of protecting the communications of our firmware update solution. To do so, we used Wireshark[1], which is a well-known protocol analyzer, to capture firmware update sessions in order to measure the amount of traffic they generated. A diagram of the set-up is present in Figure 5.2.

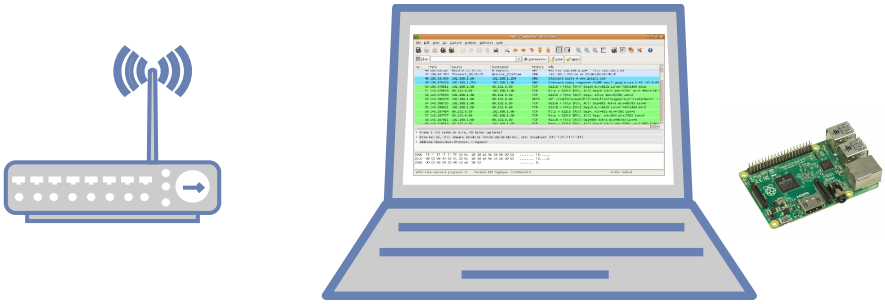

Figure 5.2: Set-up diagram of measurements of network overhead caused by UpdaThing.

We cannot directly measure the overhead of each protocol (ciper suite negotiations, the transmission of X.509 certificates, HTTP headers, etc.) because TLS connections, which are backed by ephemeral Diffie-Hellman cipher suites, prevent us from autonomously generating the shared secret derived by the Diffie-Hellman algorithm [36], even when we do have access to both servers' private RSA keys, as the keys used in the protocol are ephemeral[2]. This is as opposed to cipher suites that use shared secret keys directly derived from pre-shared secrets. Because Node.js does not allow us to generate a NSS Key Log file (a file that contains all keys and parameters used in a TLS session) consumable by Wireshark and used to decrypt TLS connections[3], only estimates from raw traffic and known facts regarding the TLS headers and messages are possible.

In Table 5.2, the readers may see the average of the measurements that were performed. For each HTTP request, three measurements were taken from the Transmission Control Protocol (TCP) conversations statistics screen of Wireshark. Both the device and a laptop acting as both the Update Server and the Signing Server were connected to the same wireless router, which may also incur in network overhead as TCP retransmissions are likely to occur. For reference, the firmware update image that was used for testing purposes had a size on disk of 140 252 672 bytes.

The discrepancy between the measured value (148 MB) and the size of the image on disk (140,25 MB) is caused by the network overhead inherent to the TLS protocol, which adds 6% [37] more of network traffic. That happens because TLS adds an entry that varies between 20 and 40 bytes in size to the header of the packet, adding to the entries of the upper protocols, such as a 20-byte header for IP, and 20-byte header for TCP (with no options). As a result, there can be 60 to 100 bytes of overhead for each

[1] `https://www.wireshark.org/`

[2] `https://wiki.openssl.org/index.php/Diffie_Hellman#Diffie-Hellman_in_SSL.2FTLS`

[3] `https://github.com/nodejs/node/issues/2363`

[3] The same image was downloaded and flashed three times. We used some database tampering to change the timestamp of the image to force its installation.

[4] The image is not transmitted. The device is purely notified and then proceeds to download the image in a separate HTTP request.

Action	Device -> Server	Server -> Device	Total
Checking for update images newer than the timestamp of the current image	3.1 kB	4.4 kB	7.5 kB
Sending of I'm Alive messages	3.1 kB	4.6 kB	7.7 kB
Downloading of firmware update image[3]	1.8 MB	146.9 MB	148.7 MB
Push notification of new firmware update image[4]	4.5 kB	3 kB	7.5 kB

Table 5.2: Network overhead for each HTTP request in bytes (1000 bytes = 1kB, etc.).

packet, which at a typical Maximum Transmission Unit (MTU) size of 1500 bytes, constitutes a minimum of 6% overhead. The authors believe the measured overhead values are negligible with modern Internet connections and are a small trade-off for the security properties guaranteed by UpdaThing.

5.5 Energy Consumption

As IoT gateways are devices that are usually running 24/7, we need to ensure that our solution is energy efficient as not to pose an obstacle for product adoption. Therefore, we went ahead and tested our Device Daemon in several states in regards to the energy consumption of the device. We placed a multimeter in series with the Raspberry Pi 2, with the Wi-FI USB adapter plugged in (TP-Link TL-WN722N), and proceeded to record the current flowing through the circuit. A diagram of such set-up can be found in Figure 5.3.

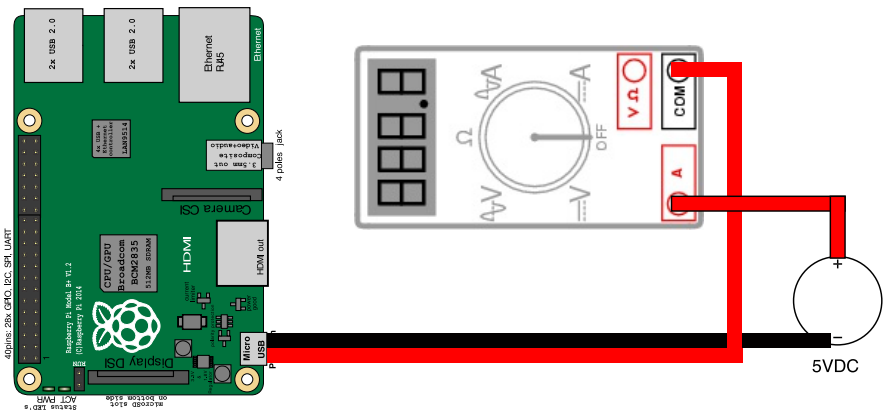

Figure 5.3: Diagram of set-up for current measurements.

As is widely known, Electric Power is calculated by $P = VI$. The Raspberry Pi 2 operates at 5V. The average of the results that we obtained are shown in Table 5.3, three measurements were taken for each state and their average was then computed.

State	Current (A)	Power (W)
Idle	0.46	2.3
Wireless Radio blinking (in operation)	0.47	2.35
Node.Js VM startup + Update Daemon checking for updates	0.52	2.6
Downloading firmware update image	0.51	2.55
Verifying firmware update image	0.50	2.5
Flashing firmware update image	0.34	1.7

Table 5.3: Device energy consumption in each state.

We should note that the difference that exists between states of regular operation and the state of flashing the firmware image to memory has to do with the powering on of the Wi-Fi adapter, which is not in operation before the device finishes booting, and that, depending on the bandwidth of the Wi-Fi connection, the time spent in the state of downloading a firmware update image varies. As can be concluded from the previous table, the difference between the device in an idle state and downloading and verifying update images is in the order of deciwatts. The authors believe such a difference is negligible and should not matter in deployment scenarios where the device is connected to a wall outlet.

Imagining a scenario where the Raspberry Pi is connected to a 3000mAh battery, a common capacity for a high-end smartphone battery, and using a factor of 0.7 to account for external factors which can affect battery life, the operating time of the device on the battery can be computed by $BatteryCapacity(Ah) \div DeviceConsumption(A) \times 0.7$. Therefore, in an idle state, the device can last for 4.57h on the previously mentioned battery. While only downloading firmware update images (doing it in a loop), it can last for 4.12h. And finally, it could last for 4.2h while verifying update images in a loop.

Since the device is in the idle state most of the time, as downloading a 120Mb image takes minutes in a broadband Internet connection, verifying it takes about 8s and flashing it to the main memory of the device takes less than a minute, taking into account booting time, all other states are negligible to its long-term power consumption. This depends, of course, on whether the main application of the device is radio intensive or not, in which case its 'idle' power consumption is relatively higher.

5.6 Cloud Deployment Cost

Evaluating our work in terms of costs it introduces when deployed using a cloud provider is a good indicator of how the product may fare in the eyes of companies' business managers. We believe that, since UpdaThing is a free and open-source system, its main users would be start-up ventures, as they are usually the ones developing IoT products from scratch and would benefit greatly from incorporating our system in them. Amazon Web Services (AWS) is one of the most popular cloud provider [4] and is widely used by start-ups as the place of their system deployments. Estimating the costs that UpdaThing would impose in AWS is, thus, a good selling point to be made to the future users of this system.

Firstly, let us estimate how much space our *I'm Alive* messages take on disk. Currently, the system stores all of these messages in a MySQL[5] database and we have to take into account the way it stores

[4] http://www.bbc.com/news/business-32442268
[5] MySQL is a relational database management system. `https://www.mysql.com/`

its data on disk. Data type sizes of Innodb (a storage engine of MySQL) are well known[6] and can be used to estimate the size of the MySQL database on disk. Our *I'm Alive* messages are kept as records in a table with the schema described in Table 5.4. The total size per record on disk is, thus, 126 bytes. Each device sends $(24 \times 60)/5 = 288$ *I'm Alive* messages per day, with a default configuration of sending a message every 5 minutes. Therefore, in a day, each device produces $288 \times 126 = 36.29kB$ of database records per day. One million devices would require 36.29GB of database records per day. AWS Elastic Block Storage (EBS) Solid State Drives (SSDs) have a price of $0.11 per GB of provisioned storage per month [7], which means that, for storing a day worth of data for one device it would cost 3.99 $\times$ 10^{-6} $ per month and for storing a day worth of data for one million of devices it would cost 3.99$ per month. In order to calculate for *n* days worth of storage, one would just compute $n \times 3.99\$$.

Column	Data type	Total size (bytes)
device_id	text (64 characters)	68
firmware_version	int(11)	4
ip	varchar(45)	46
port	int(11)	4
timestamp	timestamp	4

Table 5.4: Columns, data types and total sizes of *I'm Alive* message records.

Finally, we should estimate the costs of network transfers initiated by our system. Looking at Table 5.2, we have traffic discriminated between outgoing (OUT), from the server to the device, and incoming (IN), coming from the device to the server. At the time of writing, IN traffic (from Internet to AWS EC2) was free for all AWS deployment regions[8], however OUT traffic (AWS EC2 to Internet) was paid for in a tiered scheme. This scheme is present in Table 5.6 and allows us to compute the following estimates, $ stands for USD (United States of America Dollar):

- Checking for update images on the boot of the device produces 4.4 kB of OUT traffic which means that, for every boot of the device, it costs the manufacturer, at most, $4.4 \times 10^{-6}GB \times 0.09\$/GB = 3.96 \times 10^{-7}\$$, costing about 0.4$ for one million device boots.
- Every device sends 288 *I'm Alive* messages per day, which amounts to $288 \times 4.6kB = 0.0013248GB$, which costs $0.0013248GB \times 30 \times 0.09\$/GB = 0.00357696\$$ per device per month, meaning that, for one million of devices, the manufacturer would have a receipt for 3576.96$ at the end of the month
- Downloading update images is another beast entirely. Our measurements indicate that, for a 140MB update image, about 146.9MB is transferred from the server to device, meaning that each update costs $0.1469GB \times 0.09\$/GB = 0.013221\$$ per updated device, which may not seem like much, but quickly piles up. One million of updated devices costs, thus, 13221$. The manufacturer has to find a way to balance the costs of this updates with their frequency, while attempting to not jeopardize their devices' security.
- Push notifications are sent out to all connected devices and take about 3kB of OUT traffic. This means that, for each connected device, $3 \times 10^{-6} \times 0.09\$/GB = 2.7 \times 10^{-7}\$$ would be paid to the cloud provider, costing 0.27 $ to send a update push notification to one million of connected devices. The number of connected devices is usually lower than the number of sold devices, which makes it more affordable.

[6] http://dev.mysql.com/doc/refman/5.7/en/storage-requirements.html
[7] https://aws.amazon.com/ebs/pricing/
[8] https://aws.amazon.com/ec2/pricing/

Amount	Price
First 1 GB / month	$0
Up to 10 TB / month	$0.090 per GB
Next 40 TB / month	$0.085 per GB
Next 100 TB / month	$0.07 per GB
Next 350 TB / month	$0.05 per GB
More	Contact sales

Table 5.5: Pricing for AWS EC2 outgoing data transfers.

Lifetime costs of a device for the manufacturer encompass the cost per month of storing a given period of *I'm Alive* messages, the amount of traffic that the device generates and a shared cost for *n* devices connected to the same Update Server for its running time (EC2 instance pricing). Since we cannot estimate the latter, as it would imply an exhaustive scalability study that we chose not to pursue, and because it should be relatively low, as any robust instance type should be able to accommodate hundreds of devices, its cost per device should be negligible, when compared to the other costs.

A reasonable period of time for device renewal is 2 years. During these 2 years, a gateway, using our update system, would cost $(2 \times 12) \times (30 \times 3.99 \times 10^{-6}\$) \approx 0.003\$$ to store its *I'm Alive* messages for a period of 30 days. Assuming that the device boots up, at most, once per day, for the period of 2 years, the device would cost the manufacturer $(2 \times 12 \times 30) \times 3.96 \times 10^{-7}\$ \approx 0.0003\$$ for doing update checks on boot. Sending *I'm Alive* messages costs about 0.0036$/month. which means that the manufacturer has to pay the cloud provider, at most, for 2 years of device operation, $2 \times 12 \times 0.00357696\$ \approx 0.086\$$. It is reasonable to assume that the manufacturer would release 10 updates in the lifetime of the device of 2 years, which means that, at most, 10 push notifications would have to be sent out and 10 update images would have to be downloaded. These would cost $10 \times 2.7 \times 10^{-7} = 2.7 \times 10^{-6}\$$ and $10 \times 0.013221\$ \approx 0.13\$$, respectively. Adding all of these costs up, we arrive at the grand total of approximately 0.219$ per device for its 2 years of lifespan, which is an amount that can easily be absorbed in the manufacturer' margin.

We would like to take a moment to compare our solution with service offerings by AWS. CloudFront, CDN of Amazon, has both per request and per transferred GB pricing. Amazon prices HTTP and HTTPS requests differently: HTTPS requests cost 33%[9] more than the same requests done by HTTP, although the price per transferred GB stays the same. This means that Amazon estimates a 33% margin to account for overheads introduced by HTTPS, and, in order to use a custom X.509 certificate for these HTTPS connections, Amazon charges 600$ per month to the manufacturer for the privilege. This sort of solution would only allow for the download of update images, discarding the rest of the additional features provided by UpdaThing. All of these reasons make our system more attractive in terms of costs.

5.7 Usability

It is difficult to estimate how successful a software project will be without talking to its potential users. In this section, we describe in detail how we attempted to do just that and figure out their priorities and motivations for using a update management system like UpdaThing.

We started by devising a survey composed of 11 questions aimed at establishing who our users are: their occupations, how long they have been working for, age group and why they are interested in IoT; and whether they would be keen on using such a system, asking for their own opinion regarding

[9] https://aws.amazon.com/cloudfront/pricing/

each implemented or planned feature and possible improvements and suggestions. We later leveraged our contact networks to collect 22 anonymous responses from people with different backgrounds in the technological sector using Google Forms[10].

Our respondents were predominately students as can be seen from Figure 5.4 with a heavy interest in the development of IoT-related products. These are still our potential users as the startup ecosystem is fueled by young engineers coming out of university who would like to ship their product fast, taking into account their products' minimum viable security. We have some System Administrators and Network Engineers, whose responses give credibility to our take on the architecture of the system and its security mechanisms. We did take their age groups, gender, career years and interest in IoT, but we chose not

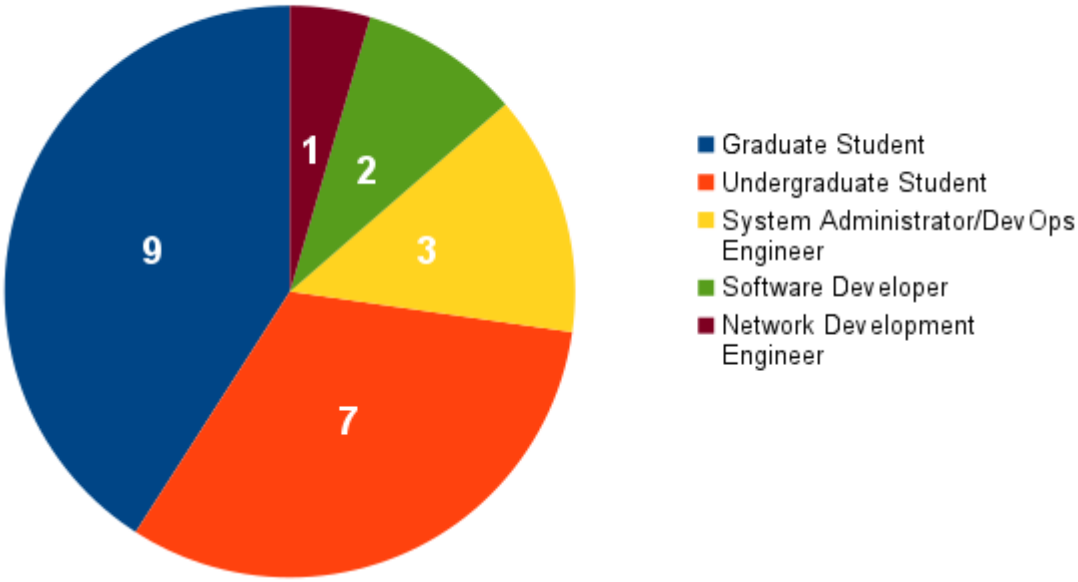

Figure 5.4: Relation between each occurrence of a occupation in the response set.

to analyze them further as we believe they do not contribute to the scientific analysis of these results, but rather to the validation of their answers, especially in regards to respondents with proven technical backgrounds.

Our survey includes a link to a brief introduction to our system present on GitHub[11], which gives an overview of its components, workflow and features, and kindly ask the respondents to take a look at it before answering the survey. Although we cannot track their behavior in this case, to understand if they thoroughly read the materials before answering, we assume they have done so and are giving rational and informed answers to the following questions.

In a 1-9 Likert scale, 1 representing that one would definitely not be interested in using Updathing and 9 representing that one would be extremely interested in doing so, we measured an average of 6.5 and a standard deviation of about 1.65, which we believe is a positive result. A representation of the answers given to this question, *"After reading our introduction page, how interested are you in using our platform for your next IoT project?"*, can be found in Figure 5.5.

We also attempted to validate our assumptions in regards to the most important features defined in the requirements in Section 1.1. Using a 1-9 Likert scale in the question "How important do you think the following features are?" and disabling repeated values for all the listed features, we effectively forced the respondents to give us both their interest in them and their relative importance. Notice that in Figure

[10] Google Forms is a Software-as-a-Service offering that allows users to develop and implement surveys and analyze their results.
[11] https://github.com/ulinux-embedded/ulinux-docs

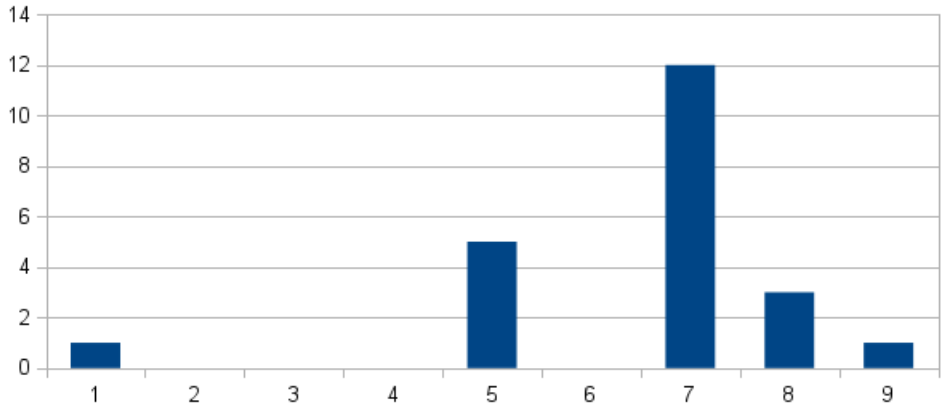

Figure 5.5: Occurrence of degree of interest in the response set.

5.6 our initial priorities held true, since the top 2 features were initially thought to be the most important and were implemented right away.

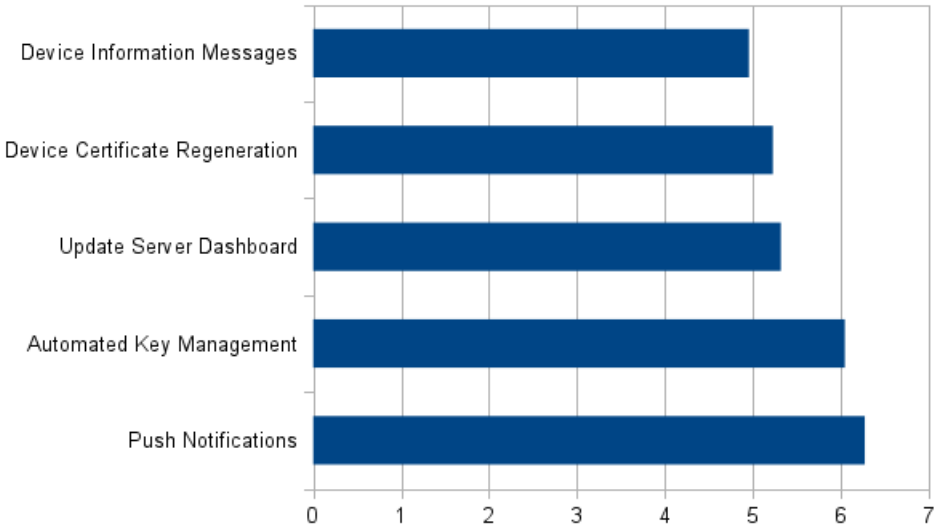

Figure 5.6: Average of degree of interest for each feature.

Even though the planned feature ”Device Information Message“ got the lower average of all of the features, we thought that including the possibility to develop an integrated logging system in future work would be interesting and a plus for manufacturers managing a large fleet of devices. Our system already implements the sending and collecting of *I'm Alive* messages that could optionally include extra useful information, such as logs, Wi-Fi network information, system uptime, etc., and we attempted to understand what kind of information the developers would like to collect by asking the respondents their interest in types of information from a predefined list and allowing them to write in their own suggestions. Figure 5.7 shows the results that we have obtained. We can clearly see that the running processes, the system uptime, Wi-Fi network information and the log of error messages of the update process would be the top 4 features to be implemented if we chose to pursue the implementation of this feature.

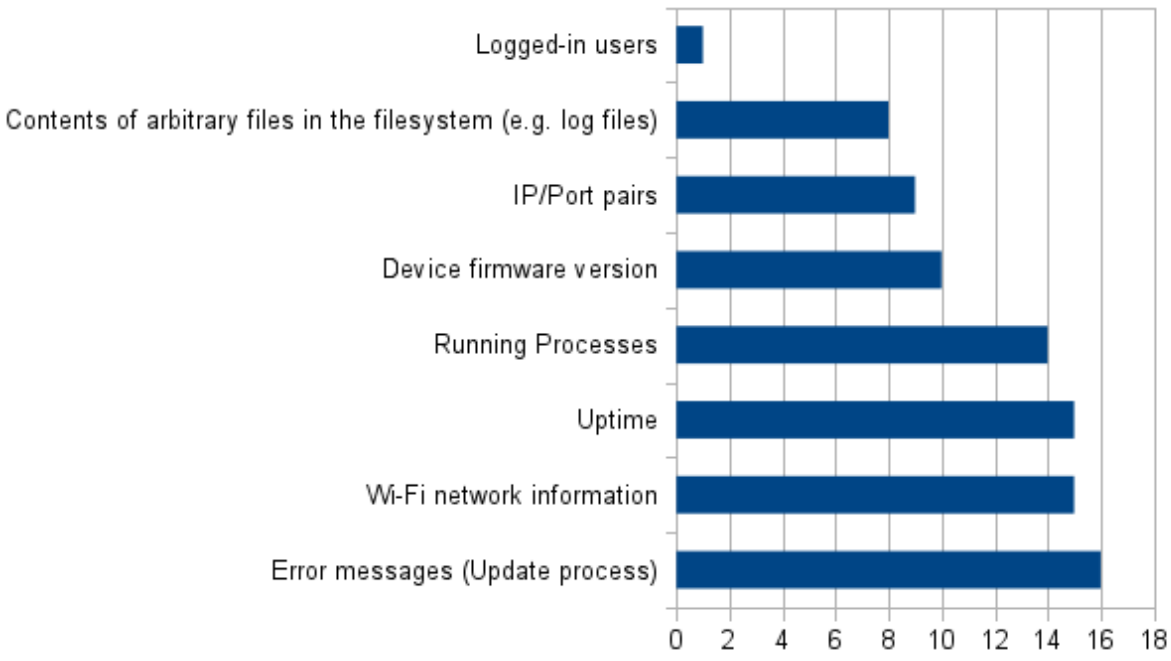

Figure 5.7: Occurrence for each kind of information in the response set.

5.8 Discussion

By performing a multifaceted evaluation to our system, we believe there are no strong obstacles for its adoption. IoT gateway devices seem to already have SoCs with enough computing power to enable the usage of Public-Key cryptography algorithms without hindering their energy consumption or taking unsurmountable amounts of time to perform cryptographic operations, which makes them already usable in production scenarios. This eases the implementation of secure communications between cheaper IoT gateways and other Internet-connected machines, being the usage of TLS therefore advised, due to the security properties that it confers and its ease of use, due to the multitude of available implementations.

On the other hand, cloud services costs are down and have lowered the barriers to entry for small ventures into the field of IoT consumer products, while also decreasing the amount of configuration and management required to run services that would otherwise require hardware, data-center space and more human resources. More competition is also entering this space, which ends up decreasing prices and increasing standards for service offerings.

The developers we have interviewed seem to be more inclined towards using open source ready-to-use solutions to solve problems they encounter, such as libraries or programming language environments, and seem to be open to relying on an open source firmware update system to manage their devices. The initial choice of using a Raspberry Pi as a development platform proved to be quite popular among tinkerers and showing the firmware update system working seamlessly on it gathered more attention than it would have if it was some other development board.

Chapter 6

Conclusion

Performing secure firmware updates is a major security challenge in the IoT as firmware update system implementations have been attacked over the years to compromise devices in bulk. We attempted to provide a solution to this problem by developing an open-source and secure firmware update system for IoT gateway devices.

By studying a body of related work, we established that, in terms of security, the system had to be capable of offering the three main properties of secure communications: Authentication, Integrity and Confidentiality. Both the update source should be authenticated, as well as the devices that attempt updates, and the firmware update image should be authenticated in its origin, by having its contents signed by who produces the software, the device manufacturer. We settled for HTTP over TLS as a way to solve our secure communication challenges and hashing and Public-Key cryptography as a means to guarantee code signing of transmitted images.

We implemented the system in Node.js, using systems such as Docker and Buildroot and libraries such as OpenSSL to help us in the development of the prototype in a timely manner, and described its architecture in this document. We then evaluated it in six aspects: how the initial requirements were satisfied, how it fared against a security analysis using the STRIDE model, measurements for network overhead and energy consumption, the associated cost of using the system in production and a gauge for the usability of the system in practice, by interviewing developers about it. By obtaining satisfiable results in all evaluation fronts, we believe that our system is usable in practice.

With this contribution to the IoT developer community, we hope to simplify the delivery of firmware updates, and to facilitate maintenance and security in the long run. We conclude that our system may be an option for manufacturers striving to implement a secure firmware update system and that opt not to develop proprietary software and infrastructure to do so.

All interested parties may find the entire source code for the project at `https://github.com/ulinux-embedded`.

6.1 Future Work

With the development of this work, we found that some further work still remains in this area: "*How to protect against attackers that have physical access to the memory of the device?*", "*What logging strategies should be implemented and used in IoT?*", "*How can this be extended to support lesser capable devices, such as sensors?*" and "*How can we make this process more efficient?*" are questions that still remain.

Since an attacker may be able to impersonate legitimate devices and interact with the Update Server

by obtaining a copy of the device CA or a device certificate, this may pose a problem when manufacturers cannot take the risk of a compromise of a single device. Further work could include the usage of secure cryptoprocessors to store keys and certificates and perform cryptographic operations in updating devices, unburdening the file system from storing these important artifacts and making it more difficult for an attacker to obtain them. In our current solution, a device impersonation attack could be mitigated by the replacement of the device CA in the next firmware update and/or the blacklisting of rogue devices by the **Update Server**, denying them any future updates. By replacing the device CA, legitimate devices could resign their certificates, and blacklisting the rogue device from updating to a firmware version that includes the new device CA would prevent its access to a legitimately signed certificate, making it impossible for the rogue device and its user to reconnect to the Update Server using the certificate signed by the old version of the device CA.

Important work should also be conducted in regards to devising lightweight logging strategies for IoT devices. Logs are important because they record relevant facts in the lifetime of the product, and about the operation of the update system. Our system produces logs that are only stored in the file system of each component and are currently not being transmitted off-site. Developing lightweight logging daemons for IoT devices that integrate with solutions like Logstash[1] would be an interesting research project.

We chose to conduct our firmware update process by replacing entire firmware images in updating devices, which presents inefficiencies when updates do not contain many changes. An interesting research project would be to extend our system with a functionality that performs differential updates for the filesystem of the device, replacing only sections that changed. Moreover, our Device Daemon could be made more efficient in its usage of CPU and RAM by being ported to a natively compiled language such as C++ or Go.

Our system was not tested or evaluated for devices with lesser capabilities, such as sensors that communicate through ZigBee or IEEE 802.15.4, and work could be pursued in this area. Updating gateway devices could cascade updates to edge/sensor devices, for instance, or take advantage of opportunistic or Delay-Tolerant Networking. UpdaThing was also not developed taking into account the maintenance of different firmware versions for devices with distinct architectures, for instance, nor is it possible to select devices for update, being the image available to all communicating devices. Implementing this sort of fine grained control would be an interesting research effort.

[1] Logstash is a data pipeline aimed at processing logs. `https://www.elastic.co/products/logstash`

Bibliography

[1] P. Suresh, J. Daniel, V. Parthasarathy, and R. Aswathy, "A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment," *Science Engineering and Management Research (ICSEMR), International Conference on*, 2014.

[2] H. Konsek, "IoT gateways and architecture," *The DZone Guide to the Internet of Things*, 2015.

[3] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[4] A. Dharmawan, "Strategies for IoT software development and delivery," *The DZone Guide to the Internet of Things*, 2015.

[5] DZone, "IoT security checklist," *The DZone Guide to the Internet of Things*, 2015.

[6] M. Reel and J. Robertson, "Firewalls and medical devices are extremely vulnerable, and everyone's pointing fingers," *Bloomberg Business*, November 2015.

[7] A. Greenberg, "Hackers remotely kill a jeep on the highway with me in it," *Wired*, July 2015.

[8] W. Stallings, *Cryptography and Network Security (4th Edition)*. Prentice Hall, 2005.

[9] IETF, "HMAC: Keyed-Hashing for Message Authentication," RFC, Network Working Group. Accessed: 2016-09-25.

[10] IETF, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) profile," RFC, Network Working Group. Accessed: 2016-09-25.

[11] ZigBee Standards Organization, "ZigBee 2007 (ZigBee Pro) Specification," 2008.

[12] N. Vidgren, K. Haataja, J. L. Patiño-Andres, J. J. Ramírez-Sanchis, and P. Toivanen, "Security threats in ZigBee-enabled systems: Vulnerability evaluation, practical experiments, countermeasures, and lessons learned," *46th Hawaii International Conference on System Sciences*, 2013.

[13] O. Olawumi, K. Haataja, M. Asikainen, N. Vidgren, and P. Toivanen, "Three practical attacks against ZigBee security: Attack scenario definitions, practical experiments, countermeasures, and lessons learned," *14th International Conference on Hybrid Intelligent Systems (HIS)*, 2014.

[14] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," *NDSS Symposium 2013*, 2013.

[15] N. Jones, "Exploiting embedded devices," *SANS Institute InfoSec Reading Room*, 2012.

[16] K.-H. Baek, S. Bratus, S. Sinclair, and S. W. Smith, "Attacking and defending networked embedded devices," *Workshop on Embedded Systems Security*, 2007.

[17] L. K. Shade, “Implementing secure remote firmware updates,” *Embedded Systems Conference Silicon Valley*, 2011.

[18] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, “A look in the mirror: attacks on package managers,” *CCS ’08 Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[19] B. M. Luettmann and A. C. Bender, “Man-in-the-middle attacks on auto-updating software,” *Bell Labs Technical Journal*, 2007.

[20] J. C. R. D. Justin Samuel, Nick Mathewson, “Survivable key compromise in software update systems,” *CCS ’10 Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[21] P. Stone and A. Chapman, “Wsuspect: Compromising the windows enterprise via windows update,” *Black Hat USA 2015*, 2015.

[22] D. K. Nilsson and U. E. Larson, “Secure firmware updates over the air in intelligent vehicles,” *ICC Workshops - 2008 IEEE International Conference on Communications Workshops*, 2008.

[23] W. Itani, A. Kayssi, and A. Chehab, “Petra: a secure and energy-efficient software update protocol for severely-constrained network devices,” *Q2SWinet ’09 Proceedings of the 5th ACM symposium on QoS and Security for Wireless and Mobile networks*, 2009.

[24] L. C. Costa, R. A. Herrero, M. G. D. Biase, R. P. Nunes, and M. K. Zuffo, “Over the air download for digital television receivers upgrade,” *Journal IEEE Transactions on Consumer Electronics archive, Volume 56 Issue 1, February 2010*, 2010.

[25] L. Katzir and I. Schwartzman, “Secure firmware updates for smart grid devices,” *Innovative Smart Grid Technologies (ISGT Europe), 2011 2nd IEEE PES International Conference and Exhibition on*, 2011.

[26] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM (CACM), Volume 13 Issue 7*, 1970.

[27] Y. Cao, C. Hu, B. Ma, H. Hao, L. Li, and W. Wang, “Secure method for software upgrades for implantable medical devices,” *Tsinghua Science and Technology (Volume:15 , Issue: 5)*, 2010.

[28] M. Garrels, “Linux - boot process, init and shutdown.” `http://www.tldp.org/LDP/intro-linux/html/sect_04_02.html`. Accessed: 2016-09-25.

[29] M. Garrels, “Linux - general overview of the linux file system.” `http://www.tldp.org/LDP/intro-linux/html/sect_03_01.html`. Accessed: 2016-09-25.

[30] Kernel Newbies, “Linux versions.” `https://kernelnewbies.org/LinuxVersions`. Accessed: 2016-09-25.

[31] R. Landley, “ramfs, rootfs and initramfs.” `https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt`. Accessed: 2016-09-25.

[32] S. Mugabi, “Initramfs tutorial.” `http://nairobi-embedded.org/initramfs_tutorial.html`. Accessed: 2016-09-25.

[33] Beyond Logic, “Understanding the raspberry pi boot process.” `http://wiki.beyondlogic.org/index.php?title=Understanding_RaspberryPi_Boot_Process`. Accessed: 2016-09-25.

[34] “Buildroot.” https://buildroot.org. Accessed: 2016-09-25.

[35] “Open Connectivity Foundation - UPnP.” http://openconnectivity.org/upnp. Accessed: 2016-09-25.

[36] W. Diffie and M. E. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, November 1976.

[37] I. Grigorik, *High Performance Browser Networking*. O’Reilly, 2013.