UpdaThing: a secure firmware update system for Internet of Things devices

Tomás Alexandre Diniz de Pinho
Instituto Superior Técnico, Universidade de Lisboa
Av. Rovisco Pais, 1
1049-001 Lisboa, Portugal
tomas.pinho@tecnico.ulisboa.pt

ABSTRACT

The Internet of Things (IoT) has amassed interest from industry and consumers, and devices belonging to this class are being deployed at a rapid rate, to collected data that will help to solve problems in many different domains. Manufacturers have to provide support to such deployments and thus have to patch the software running on these devices when a bug is found and corrected. Therefore, updating the firmware running on these devices requires a firmware update mechanism, which presents security challenges, as it can be used by malicious actors to compromise these devices in bulk.

In this paper, we propose, implement and evaluate a firmware update system that is easy to deploy and easy to use, assures code authenticity (integrity and proof of origin), confidentiality and freshness in the update process, defeating the most relevant external security threats.

Keywords

Internet of Things; Firmware; Software update; Over-the-air update; Security; Public Key Infrastructure; Embedded Systems

1. INTRODUCTION

For the past few years, there has been a surge in popularity of the Internet of Things (IoT) among consumers and manufacturers and it is now a well-known class of devices with multiple software libraries and development kits available[25]. IoT systems, in small-business or home deployments, are usually implemented via a set of sensors that capture events or measurements; and a central IoT gateway, a more capable device that sits between resource-constrained devices, which operate on specific protocols, and the Internet, serving as a bridge between them[17]. The sensors transmit data using specific low-energy protocols to the IoT gateway which may pre-process the data or add some more data collected by its own sensors (if it has them). The gateway then sends collected data to a Data Center, where it may be processed and made available through some other media (i.e. Web/Mobile Applications). The role of the gateway is illustrated in Figure 1.

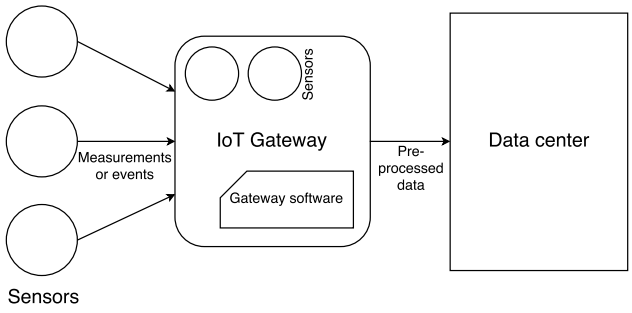

Figure 1: Gateway connecting limited sensor devices to a Data Center.

As the IoT continues to grow, so does the number of connected devices that require manufacturer support. "All software has bugs" is a known fact of Software Engineering [19], so manufacturers have to plan for and provide some sort of patch management process that corrects them in end-user devices.

Updating software in IoT devices is usually performed via updating their firmware in its entirety[5]. This process involves some firmware update mechanism that allows devices to connect to a server deployed by the manufacturer in order to download and update to a newer firmware image. However, if not correctly designed, this mechanism may pose a severe threat as it may be used by malicious actors to compromise these devices in bulk, obtain proprietary code or sensitive information[24].

This paper presents an open-source secure firmware update mechanism aimed at IoT gateways. Our system assures proof of origin, integrity and in-transit confidentiality of the update images, while still being easy to use and deploy.

We assess the security of our system by conducting a STRIDE threat assessment on its endpoints, measuring network overhead and energy consumption, before comparing it to relevant related work.

1.1 Overview

This paper is organized as follows: we present all necessary background on the development and updating of firmware for embedded systems in Section 2. In Section 3, we specify the requirements that our system must satisfy and the mechanisms which we will be using to provide our security guarantees. In Section 4, we provide the implementation details of our system, what components it has and how they

interoperate. In Section 5, we evaluate our system in five domains: which initial requirements it satisfies, how it fares against a STRIDE analysis, which includes possible attacks and their mitigation, how much network overhead it causes, how much energy it consumes and how much it costs to run it. In Section 6, we take a look at previously studied related work and briefly analyse some of its shortcomings and how our system fares against it, comparatively. Finally, in Section 7, we conclude this document and propose further work in this field.

2. BACKGROUND

In this section we provide technical details for the firmware image operations that are relevant to our solution.

2.1 Image Flashing

IoT devices possess some kind of permanent bootable memory from which the device boots its Operating System (OS) via its bootloader[8]. In order for the device to be updatable after being initially flashed at a factory plant, this memory must be writable by the device's bootloader, or even by its OS. The manufacturer has to write some specialized code, that is run alongside the device's main application, which in turn is responsible for looking for and fetching new firmware update images. The code may or may not perform additional functionality (such as image verification) and flash the image directly or instruct the bootloader to do so on the device's next reboot.

Devices running the Linux OS may write to their permanent bootable memory (their root filesystem) as long as its not being used by any running services or applications. The manufacturers usually implement some sort of two-phased boot process in which a minimal *initramfs*[18][1] is loaded into the device's Random Access Memory (RAM)[20] that can perform this sort of operations on the permanent bootable memory before booting the entire system, with its services and applications.

2.2 Image Verification

After downloading an update image, the device's firmware update mechanism may perform some steps to verify the transferred image. Some devices perform integrity checks, by running an hashing function (i.e., CRC32, MD5, SHA1/2, etc.) through the image and comparing its output to additional downloaded metadata[7]. This sort of integrity check does not provide any security guarantees, as the metadata and image could have been altered in-transit by a malicious actor, but guarantees some safety in the update process by safeguarding against data corruption of the update image.

Other devices may perform proof of origin checks of the downloaded update images by coupling the previously mentioned integrity checks with a signed hash of the update image[21][11]. These devices calculate the hash of the update image and then perform a cryptographic operation on it to verify that a specific shared-key or public-key has been used to sign the downloaded hash, present in the update metadata, thus verifying the image's origin. We should note that shared-keys are susceptible to attacks since an attacker may be able to obtain it by exploiting a device and then using it to sign images for all devices of the same model. Public-Private Key pairs are more difficult to obtain since it is a difficult process to obtain a Private Key (held by the manufacturer) from a Public Key (present in devices)[23].

[1]The *initramfs* is a cpio ("copy in and out") archive of the initial file system that gets loaded into memory during the Linux startup process. It solves the "chicken or the egg" problem of loading required modules for using file systems before actually reading the kernel image inside the same file systems.

2.3 Image Generation

In order to have an update image to flash a device with, it must be generated somehow. A filesystem with the OS, required services and applications must be put together in order to produce a flashable image. Linux-based devices must, at the bare minimum, run a Linux kernel, an init system (a program that is responsible for launching all services and applications after the kernel finishes booting) and required services and applications[10]. Therefore, the manufacturer has to configure and compile the Linux kernel with any required kernel modules for the target architecture, write or choose an init system and write or choose required services and applications to include in the image. Usually, this process is automated by the development or usage of scripts that do all these tasks in a repeatable manner to integrate all manufacturer-developed applications into a useful filesystem image.

3. TECHNICAL SPECIFICATION

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. These devices may be powered, for instance, by ARMv6 and ARMv7-based SoC packages. As all Raspberry Pi[2] 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.

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 an IoT gateway with included sensors.

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

[2]The Raspberry Pi is a series of credit card–sized single-board computers developed in the United Kingdom by the Raspberry Pi Foundation.

3.1 Requirements

As with any software project, we had to carefully define the requirements for our system. As our system aims at solving specific security problems that concern a firmware update image, the majority of such requirements would be to provide certain security guarantees. Therefore, our solution aims to satisfy the following requirements:

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

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

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

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

3.2 Mechanisms to guarantee security

Transport Layer Security (TLS) sockets were chosen to satisfy Requirement 1 and 2, via the usage of the HTTPS protocol for all of the communications. The certificate validation phase of the TLS handshake can validate the identities of all entities participating in the update process and in-transit integrity and confidentiality are guaranteed by the cipher suites negotiated by TLS[13]. Both Server and Client certificates shall be used to identify all entities, namely the Server certificates for both the Update Server and Signing Server and the Client certificates for the devices. The Servers' Certificate Authority (CA) and the Clients' CA should not be the same, since the clients should be capable of generating and signing their own certificates, by having a local copy of the CA.

Additionally, in order to guarantee proof of origin, the system can verify the integrity of the downloaded image after transferring it by calculating a hash using SHA512 of the image and verifying its signature using a trusted RSA Public-Key.

4. UpdaThing

UpdaThing is the name of this project and is an umbrella term for all its components. UpdaThing has a Linux distribution and its development tools, two distinct server implementations and a device daemon. Figure 2 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.

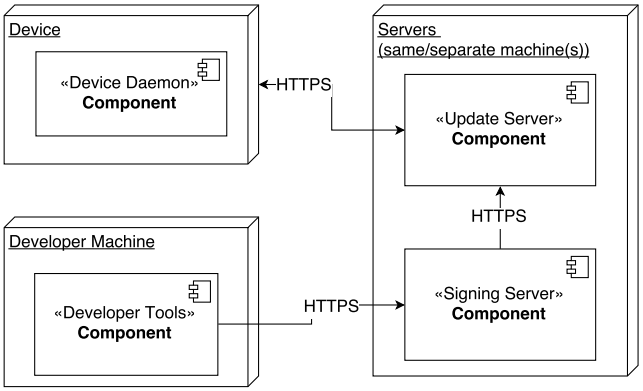

Figure 2: UpdaThing UML Deployment Diagram

4.1 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 device's boot process and 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[1]. 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 second partition's contents. 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 initiating Busybox's init system.

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. 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.

4.2 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 filesystem images. An included tool may be used to upload the resulting images to the Signing Server.

The Signing Server is responsible for receiving raw firmware update images 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[3] container that runs a custom shell script which in turn mounts the filesystem image and copies the required files onto it. Then, 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[4] package along with the image. Finally, the Signing Server uploads this package to the Update Server.

The Update Server is responsible for notifying alive clients of new updates, serving update packages by their ID when requested, receiving I'm Alive messages from devices and responding to queries for newest update packages' IDs by timestamp.

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 device's boot 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 the Signing Server's Signing Public-Key that is installed on the device's permanent memory. The Device Daemon also attempts to establish port forward mappings with the router it is connecting to through the usage of UPnP[2]. This enables the Device Daemon to receive push notifications from the Update Server Directly.

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 HyperText Transfer Protocol (HTTP), TLS and a Cryptography library (based on OpenSSL bindings) that more than fits our use case. Moreover, Node.js's vibrant open-source community 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.

4.3 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 3. 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 Update Server's endpoint "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 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.

5. EVALUATION

5.1 Requirements Satisfaction

In this section we give a brief summary of what requirements were satisfied and how.

Requirement 1 was satisfied by the usage of X.509 certificates signed by trusted Certificate Authorities in the authentication part of the TLS protocol of HTTPS. Requirement 2 was satisfied by the use of TLS which guarantees confidentiality and integrity of transferred data. Requirement 3 was satisfied by the Development Tools that were developed as part of this system that allow the developers to automatically configure and generate complete Linux filesystem images using Buildroot. 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.

[3]Docker is an open-source project that automates the deployment of applications inside software containers.

[4]Tar stands for Tape ARchive and it is a file archiving format.

5.2 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 3 we present a DFD for all UpdaThing's components 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 in Table 1.

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

5.3 Network Overhead

We measured the network overhead of protecting the communications of our firmware update solution. To do so, we used Wireshark, which is a well-known protocol analyzer, to capture firmware update sessions in order to measure the amount of traffic they generated. Unfortunately, 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[5]. Since TLS connections, which are backed by ephermeral Diffie-Hellman cipher suites, prevent us from autonomously generating the shared secret derived by the Diffie-Hellman algorithm[9], even when we do have access to both servers' private RSA keys, as the keys used in the protocol are ephemeral[6], we cannot directly measure the overhead of each protocol (ciper suite negotiations, the transmission of X.509 certificates, HTTP headers, etc.). This is as opposed to cipher suites that use shared secret keys directly derived from pre-shared secrets. Therefore, only estimates from raw traffic and known facts regarding the HTTP headers and payload (request body) are possible.

In Table 2, the readers may see the average of the measurements that were performed. For each HTTP request, three measurements were taken from Wireshark's Transmission Control Protocol (TCP) conversations statistics screen. 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.

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

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

Action	Amount of traffic generated (bytes)
Checking for update images newer than the current image's timestamp	7.6 kB
Sending of I'm Alive messages	7.6 kB
Downloading of firmware update image[3]	148 MB
Push notification of new firmware update image[4]	4.6 kB

Table 2: Network overhead for each HTTP request

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 atleast 6%[12] 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 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.

5.4 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 device's energy consumption. We placed a multimeter in series with the Raspberry Pi and proceeded to record the current flowing through the circuit. A diagram of such set-up can be found in Figure 4.

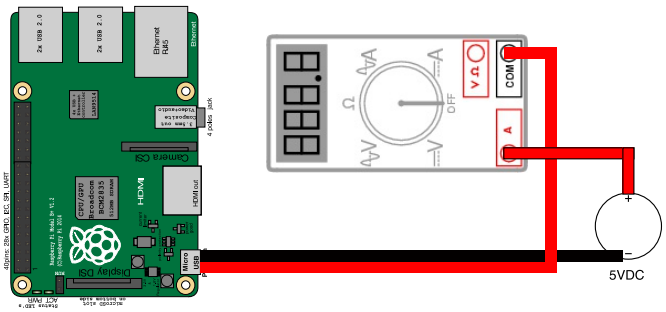

Figure 4: Diagram of set-up for current measurements

As is widely known, the formula for the Electric Power is $P = V\ I$. Since the Raspberry Pi 2 operates at 5V, the results shown in Table 3 can be obtained.

[3]The same image was downloaded and flashed three times. We used some database tampering to change the image's timestamp 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.

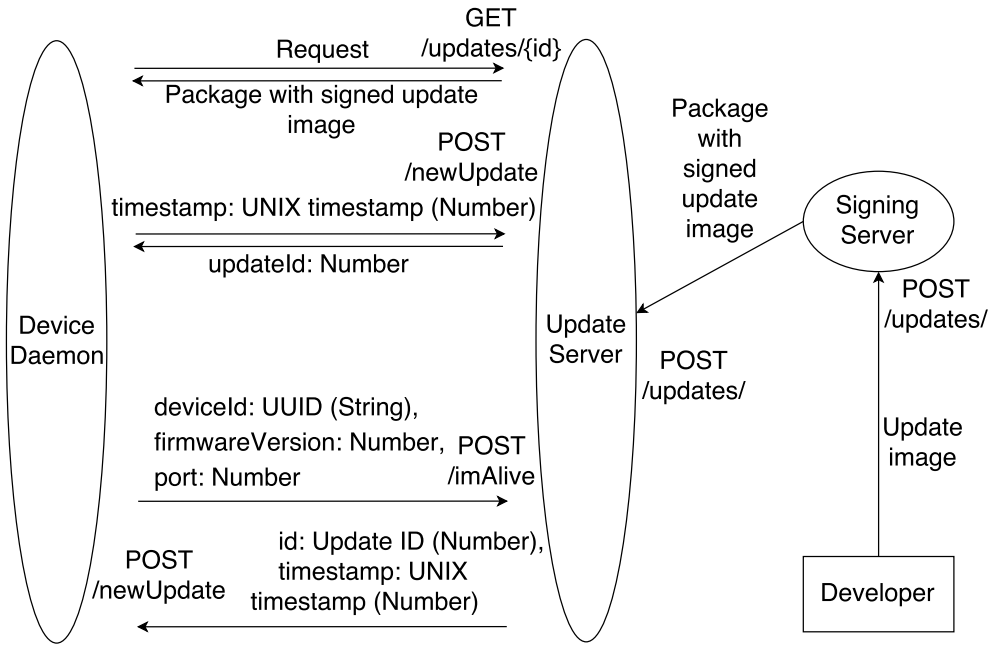

Figure 3: Data Flow Diagram of UpdaThing's components

Endpoint	Server	Client	Possible attacks
GET /updates/{id}	Update Server	Device	Device controlled by an attacker could download update images. Malicious actor could perform a Man-in-the-Middle (MitM) attack and force a device to download a given update image (could cause a Denial of Service (DoS), achieve Remote Code Execution (RCE) if the signing server's key 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 client's update image, **Information disclosure** of firmware update image and **Elevation of privilege** in regards to access to the updating device are, thus, possible.
POST /newUpdate	Update Server	Device	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 Update Server's identity and **Tampering** with the update ID are possible.
POST /updates/	Update Server	Signing 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	Update Server	Device	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/	Signing Server	Developer	Threat actor could upload malicious images to get them signed and installed on devices, or sniff unsigned legitimate firmware images. **Spoofing** a developer's or Signing Server identities, **Tampering** with installable firmware update images, **Repudiation** of uploaded images and **Information Disclosure** of firmware update images are possible.
POST /newUpdate	Device	Update Server	Attacker could pose as the Update Server and notify the device of an outdated update image, forcing it to rollback its software version. **Spoofing** Update Server's identity and **Tampering** with installed firmware version are possible.

Table 1: STRIDE analysis for each endpoint

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 3: Device energy consumption in each state

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 device's operating time 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 4.12h, and could last 4.2h verifying update images.

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 device's main memory 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 device's main application is radio intensive or not, in which case its 'idle' power consumption is relatively higher.

5.5 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. Amazon Web Services (AWS) is one of, if not the most, popular cloud provider [7] and is widely used by start-ups as the place of their system deployments.

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[8] database and we have to take into account the way it stores its data on disk. Innodb's (MySQL's storage engine) data type sizes are well known[9] 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 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 [10], which means that, for storing a day worth of data for one device it would cost 3.99×10^{-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 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 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[11], however OUT traffic (AWS EC2 to Internet) was paid for in a tiered scheme. We can compute the following estimates:

- Checking for update images on the device's boot produces 4.4 kB of OUT traffic which means that, for every device's boot, 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

[7] http://www.bbc.com/news/business-32442268

[8] MySQL is a relational database management system. https://www.mysql.com/

[9] http://dev.mysql.com/doc/refman/5.7/en/storage-requirements.html

[10] https://aws.amazon.com/ebs/pricing/

[11] https://aws.amazon.com/ec2/pricing/

of connected devices. The number of connected devices is usually lower than the number of sold devices, which makes it more affordable.

A device's lifetime costs 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 device's lifetime 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, Amazon's Content Delivery Network (CDN), has both per request and per transferred GB pricing. Amazon prices HTTP and HTTPS requests differently: HTTPS requests cost 33%[12] 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.

6. RELATED WORK

In preparation for this work, the authors studied available research on firmware update systems. In this section we make a brief analysis on the studied systems and how they compare against UpdaThing. You may find a summary of the security properties guaranteed by such systems in Table 5.

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 do not take into account code signing that happens in some systems via symmetric keys. Confidentiality of the downloaded firmware image is commonly guaranteed through the usage of image-encrypting symmetric keys.

CAO et al's system[4] 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 IPG's controller or between the IPG's controller 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.

As can be seen by analyzing table, we could not find a fully implemented research system that guaranteed all four properties without leaving room for a successful attack. For instance, Nilsson and Larson's[22] system leaves the key management to further work and works with unique permanent pre-shared keys between vehicles and the portal, which makes the system easier to attack, in regards to the firmware image confidentiality it guarantees.

Two proprietary systems closer to what we developed are the NEST thermostat[21] and the Google OnHub router[11]. They both update via HTTPS and make use of TLS as a way to assure data integrity and confidentiality while in transit, and both make use of Public-key Cryptography for code signing.

7. CONCLUSIONS

With the work described in this paper we propose a solution for a major security problem in IoT: how to perform secure firmware updates in gateway devices. We established requirements for a system attempting to solve this issue and chose HTTP over TLS as a way to solve our secure communication challenges. In order to guarantee code signing of transmitted images, we used Binary Hashing and Public-Key cryptography. We implemented the system, described its architecture and evaluated it in three aspects: a security analysis, measurements for network overhead and energy consumption, and we finished by comparing it to existing systems.

Further work that may be done in this area includes the usage of Trusted Platform Modules 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. Important work should also be conducted in regards to devising lightweight logging strategies for IoT devices. Our system produces logs that are only stored in each component's file system and are currently not being transmitted off-site. Developing lightweight logging daemons for IoT devices that integrate with solutions like Logstash[13] 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

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

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

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.[6]	✓	✓	✗	✓	asymmetric
	Itani et al.[15]	✓	✓	✗	✗	symmetric
	Katzir and Schwartzman[16]	✓	-	-	-	-
	CAO et al.[4]	✓	✓	✓	✗	symmetric / none
Proprietary	Home routers[14]	✗	✗	✗	✗	✗
	Android OS[3]	✓	✓	✗	✓	asymmetric
	NEST thermostat[21]	✓	✓	✓	✓	asymmetric
	Google OnHub router[11]	✓	✓	✓	✓	asymmetric

Table 5: Summary of guarantees provided by systems in related works

changes. An interesting research project would be to extend our system with a functionality that performs differential updates for the device's filesystem, replacing only sections that changed.

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 in the *tar.gz* hosted at http://bit.ly/28lpB9I.

8. REFERENCES

[1] Buildroot. https://buildroot.org.

[2] Open Connectivity Foundation - UPnP. http://openconnectivity.org/upnp.

[3] Android Open Source Project. Ota updates. https://source.android.com/devices/tech/ota/index.html, 11 2015.

[4] 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.

[5] L. Case. How to update firmware on pcs and peripherals. http://www.pcadvisor.co.uk/how-to/software/how-to-update-firmware-on-pcs-and-peripherals-3237825/.

[6] 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.

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

[8] T. Davis and D. Durlin. Bootloaders 101: making your embedded design future proof. http://www.embedded.com/design/prototyping-and-development/4410233/Bootloaders-101--making-your-embedded-design-future-proof.

[9] W. DIFFIE and M. E. HELLMAN. New directions in cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, IT-22(6):644–654, November 1976.

[10] M. Garrels. Linux - boot process, init and shutdown. http://www.tldp.org/LDP/intro-linux/html/sect_04_02.html.

[11] Google. Onhub security features. https://support.google.com/onhub/answer/6309220?hl=en.

[12] I. Grigorik. *High Performance Browser Networking*. O'Reilly, 2013.

[13] IETF. The Transport Layer Security (TLS) protocol version 1.2.

[14] Independent Security Evaluators. Exploiting SOHO routers. http://www.securityevaluators.com/knowledge/case_studies/routers/soho_router_hacks.php, 2013.

[15] 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.

[16] 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.

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

[18] R. Landley. ramfs, rootfs and initramfs. https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt.

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

[20] S. Mugabi. Initramfs tutorial. http://nairobi-embedded.org/initramfs_tutorial.html.

[21] Nest. Keeping data safe at nest. https://nest.com/security/.

[22] D. K. Nilsson and U. E. Larson. Secure firmware updates over the air in intelligent vehicles. *Communications Workshops, 2008. ICC Workshops '08. IEEE International Conference on*, 2008.

[23] R. L. Rivest, A. Shamir, and L. Adleman. A method

for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2), Feb. 1978.

[24] N. J. Rubenking. Securing the internet of things. http://www.pcmag.com/article2/0,2817,2483635,00.asp.

[25] 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.