An Over the Air Update Mechanism for ESP8266 Microcontrollers

Dustin Frisch*, Sven Reißmann†, Christian Pape*
*Department of Applied Computer Science
Fulda University of Applied Sciences, Fulda, Germany
Email: {dustin.frisch, christian.pape}@cs.hs-fulda.de
†Datacenter
Fulda University of Applied Sciences, Fulda, Germany
Email: sven.reissmann@rz.hs-fulda.de

***Abstract*—Over the last years, a rapidly growing number of IoT devices is found on the market, especially in the area of the so-called smart home. These devices, which are deployed in vast numbers, are frequently in use over many years. They pose a risk to the users privacy and to the internet as a whole if not provided regularly with security patches. Hence, a fully automated process for large-scale software updates of such embedded devices must be considered. In this article, we present an implementation of a durable and stable system for building and publishing cryptographically secure firmware updates for embedded devices based on *ESP8266* microcontrollers. This includes mechanisms to build the updates from source and automatically sign, distribute and install them on the target devices.**

Keywords–IoT; Secure Updates; Over the Air; ESP8266.

I. Introduction

In todays marketplace, an explosive growth can be observed in the area of so-called smart devices, often referred to as Internet of Things (IoT). Conventional devices (e.g., door locks, light bulbs, washing machines) are extended with smart functions for remote control and monitoring. To implement the additional smart functions, small embedded computer systems are getting integrated into the devices, allowing them to connect to the local WiFi network.

In embedded systems, the software, also known as firmware, is an essential part of the system. On one side, it interacts with the hardware in a system specific way by implementing the specifications required by the components used in the system. On the other side, it provides use-case dependent functionality in interaction with general purpose hardware components. Embedded systems are often thought as systems that never change their requirements or functionality. However, practical use shows that the environment in which these systems run does, in fact, change. These changes include, but are not limited to, modifications to the expected behavior or additions to it, reconfiguration of parameters related to the communication with other systems or the users, as well as correcting errors, particularly security related issues, that have been reported after deployment and roll-out. In almost all cases, the requirements can be accomplished by changing the firmware and do not need any modification to the hardware. For updating the firmware on a system being deployed, the system must provide an interface for altering the firmware. In addition, such an interface should provide mechanisms to check which firmware is currently installed and which configuration parameters are used.

Even if systems are equipped with an interface for applying updates, the maintenance cost can still be enormous if an administrator has to interact with each device physically and the systems are located in areas where reachability is limited. If a system is already able to communicate over a network interface, this can be leveraged to apply updates on these system - this is typically referred to as *Over the Air (OTA)*. By reusing the existing communication channels, the dedicated update interface can be omitted, which leads to smaller packaging and reduces production cost. It also decreases the maintenance cost drastically, because updates can be triggered remotely. *OTA* updates enable administrators to apply automation methods on the update process allowing to roll out new releases and fixes in a controlled fashion. As an example, updates can be done on test-devices first, followed by security-critical deployments and subordinate ones can be delayed to times when the device is not utilized. Further, a feedback channel, which provides information about the update status of a devices allows administrators to apply monitoring techniques ensuring all updates are installed and devices are in the desired state.

The remaining part of this paper is laid out as follows. Section II discusses related work. Next, in Section III we present the environment, our research is based on, while Section IV defines the requirements for the implementation of an *OTA* update mechanism in this environment. A concept for the implementation is presented in Section V and a reference implementation can be found in Section VI. Finally, a conclusion and future work can be found in Section VII.

II. Related Work

Wireless sensor and actor networks are a crucial elements of today's effort to support and implement *Industry 4.0* architectures and modern manufacturing processes. Small programmable logic controllers (PLC) and cloud computing are enabler but also drivers of these new manufacturing paradigms[1]. Thus, the networked interconnection of everyday objects, the automation of home appliances and environmental metering and monitoring based on sensor and actor networks controlled by ESP-based chipsets are subject of current research. In [2], a low-cost multipurpose wireless sensor network using *ESP8266* PLCs is introduced. The usage of *ESP8266* PLCs in combination with Raspberry PI acting as base station for the sensors is discussed in [3]. The article [4] presents a home automation solution based on a *MQTT* message queue with *ESP8266*-based sensors and actors. The control of smart bulbs with PLCs is summarized in [5]. Unfortunately, soft ware update mechanisms are not addressed in these publications. The importance of regular security updates for today's infrastructures is summarized in [6]. An approach of decentralized software updates in Contiki-

 ISBN: 978-1-61208-591-3

based IoT environments are introduced in [7]. In [8], a software update solution for devices able to execute a Java Virtual Machine (JVM) is introduced. Both solutions are not applicable for small MCU devices. In [9], a diagnoses and update system for embedded software of electronics control units in vehicles is introduced. Secure firmware updates targeted for the automotive industry is introduced in [10]. Furthermore, a secure The *Over the Air* programming capabilities of the *ESP8266* PLCs are described in [11].

III. Environment

The research presented in this paper was mainly driven by *Magrathea Laboratories e.V.* [12], the local hackerspace in Fulda, Germany, in cooperation with researchers at the department for computer science at Fulda University of Applied Sciences. Requirements were clearly defined by Magrathea Laboratories' demands to provide local and remote control over various sensors and actors in the foundations rooms to visitors and members. Such components include door sensors, power sockets, temperature sensors, projectors and screens who are all managed by a home-automation controller, which is driven by the software *home-assistant* [13]. It provides direct control over all existing components using a web-based user interface and allows to define rules and automations on how these components interact.

For the component's hardware, boards based on the *ESP8266* [14] micro-controller are used. These boards feature a small and robust design, achieve very low power consumption and integrate WiFi without requiring any extra components. It integrates a Tensilica L106 32-bit micro controller unit (MCU) with a maximum CPU performance of 160 MHz, 64 kB instruction memory and another 96 kB of main memory. According to the manufacturer, the ESP8266 is among the most integrated WiFi-capable chips in the industry. While at the beginning of this research, mostly *ESP-01s* [15] boards in combination with self-developed power supplies and use-case specific hardware components were deployed, *Sonoff* [16] wireless smart switches product series offered by *ITEAD* have been integrated quickly.

The firmware for all of the *ESP8266*-based devices in the hackerspace has been implemented using a common software platform, referred to as *ESPer*. *Sming* [17], which in turn is based on the open-source software development kit (SDK) for *ESP8266*, provides the base library for this framework. It integrates a lot of other software components and provides all kinds of functionality shared by all devices, allowing to reuse parts of the source code in multiple devices.

For communication with the controller, the *Message Queue Telemetry Transport (MQTT)* [18] protocol is used. It provides a lightweight messaging mechanism implementing the publish-subscribe pattern that allows devices to listen for commands and publish their current state to the controller and other interested parties. The controller software has out-of-the-box support for this protocol, which allows easy integration of all different device types using the same patterns.

The components all share the same configuration in regard to the network access and the controller to communicate with. The configuration is provided during build time, which eschews the need for a configuration interface and reduces the management overhead, thus minimizing security leaks.

IV. Requirements

For the implementation of an OTA update mechanism, the following requirements were defined.

1) The systems must be able to perform updates on the release of new software without manual interaction. If a new firmware version is published for a type of devices, the target devices must fetch and install the new software version automatically, and start using it subsequently if no errors have occurred during the update.

2) To ensure minimal maintenance effort, the update process should be insusceptible to errors as much as possible. Even if the installation of an update fails while reprogramming the device, the system should continue to work fully functional immediately and after reboot.

3) Firmware downloads must be possible over the same WiFi connection as used during normal operation. Fetching the firmware should be done side-by-side with operational traffic.

4) The update process must be possible over any untrusted wireless network or Internet connection. To prevent possible attackers from injecting malicious software into the embedded devices, a cryptographic signature mechanism must be implemented. New firmware only gets accepted by the device, if the cryptographic signature of the downloaded firmware image can be verified.

5) To reduce network load and aim for the maximum possible uptime of the device, the update process should only be done if a new firmware version is available. In contrast, on the release of new firmware, the roll-out to all devices should be performed as fast as possible. While checking for available updates and downloading such an update, the device should continue to work as usual.

6) For easy maintenance and monitoring, each device must provide information about the currently installed firmware version and other details relevant for the update process.

7) Devices are categorized by types. Each type runs the same software and therefore provides the same functionality. As the device type is hardly coupled to the hardware and the software interacts with it on a specific way, the update process must ensure that the correct firmware is used while reprogramming.

V. Concept for implementing *OTA* updates

To implement *OTA* updates under the given requirements, we first define a topology that integrates our build infrastructure, firmware repository, and controller with the IoT WiFi network, which the devices are connected to. For our reference implementation, we particularly chose lightweight and common software projects to allow for easy exchangeability of the individual components. The base topology, as well as the specific components used is shown in Figure 1.

The source code of the *ESPer* project is published into a *Git* [19] source code repository. From there, the continuous integration (CI) system is responsible for automatically building and publishing the firmware image files, as soon as updated source code is available. It is also in charge of assembling and publishing meta-information consisting of version number and cryptographic signature required for the update process. The CI systems is described in detail in the following section. Updates to the devices firmware are either triggered actively (i.e., manual or by the CI) or on a regular schedule by the devices themselves. This process is described in Section V-B.

 ISBN: 978-1-61208-591-3

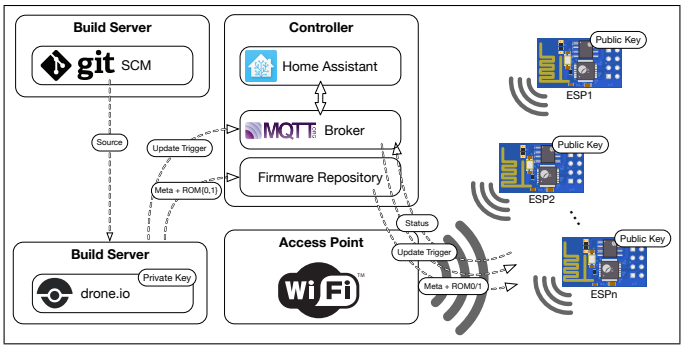

Figure 1. The base network topology.

For monitoring and maintenance purposes, each device publishes a set of information to a well-known *MQTT* topic after connecting to the network. Beside data like device type, chip and flash ID, the published data includes details about the bootloader, SDK and firmware version as well as relevant details from the bootloader configuration, like the currently booted ROM slot and the default ROM slot to boot from. This allows administrators to find devices with outdated bootloaders and helps to find missing or failed updates.

A. Common framework and build infrastructure

The framework includes a build system, which allows to configure basic parameters for all devices, including, but not limited to, the WiFi access parameters, the *MQTT* connection settings and the updater URLs. Each device requires to have the `UPDATE_URL` option set to make the update work. Skipping the option results in the exclusion of the code for update management during the build. By sharing the same code, all devices ensure to have a common behavior when it comes to reporting the device status or interacting with the home-automation controller. This eases configuration and allows to collect information about all devices at a central location.

As development on the devices usually happens in cycles, some of the projects would miss updates of the framework and therefor would not benefit from newly added features or fixed problems. Regularly updating the framework version and rebuilding the firmware would often result in an easy gain of these benefits, but requires manual interaction. Further, problems could arise if the application programming interface (API) of the framework changes. In this situation, the device firmware must be updated to use the changed API, which can be an unpleasant and complex task that leads to higher latency for firmware updates. To prevent these problems, the firmware of all devices in the hackerspace is integrated together with the framework into a larger project. By doing so, any device specific code is always linked to the latest version of the framework. The according device type is provided as a string through a global constant at compile time and it must never be changed during operation. Device specific code is organized in a sub-folder for each device type. To build the software, a *Makefile* [20] is used, which provides a simple way for reproducible builds. Whenever a new build is started, the build system scans for all device specific folders and calls the build process for each of them. After the build of the firmware has finished, the build system also creates a file for each device type, containing the build version and cryptographic signatures of the corresponding firmware images. To avoid interferences between different build environments, and to roll out new versions as quickly as possible, the code has been integrated into a CI system, which is also responsible for publishing the resulting firmware images to the firmware server queried during updates, and for notifying the devices to check for an update.

B. Device setup and flash layout

Microcontroller boards based on the *ESP8266* MCU are mostly following the same layout: the MCU is attached to a flash chip, which contains the bootloader, firmware and other application data. The memory mapping mechanism of the MCU allows only a single page of 1 MB of flash to be mapped at the same time [21] and the selected range must be aligned to 1 MB blocks.

As the firmware image to download and install possibly exceeds the size of free memory heap space, the received data must be written to flash directly. In contrast, executing the code from the memory mapped flash while writing the same area with the downloaded update leads to unexpected behavior, as the executed code changes immediately to the updated one. To avoid this, the flash is split into half to contain two firmware ROM slots with different versions, one being executed and one which is being downloaded (see Figure 2). In addition to the two firmware ROM slots, the flash provides room for the bootloader and its configuration. For alignment and easy debugging, the second block is shifted by the same amount of bytes as the first block. The gap of 8192 bytes is available to applications to store data, which can persist over application updates.

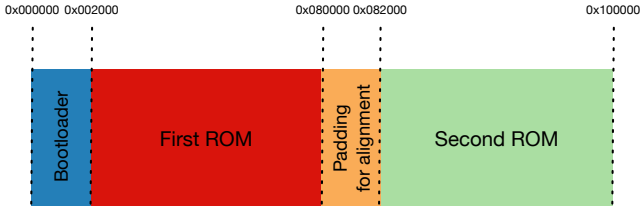

Figure 2. The flash layout used for two ROM slots.

This standby ROM slot also acts as a safety mechanism if the download fails or is interrupted as the previous version stays intact and can still be used (refer to requirement IV-2).

C. Cryptographically securing the firmware update

To ensure only valid firmware is running on the devices, a cryptographic signature of the firmware images is calculated and checked as part of the update process. For calculating and verifying the signatures of a firmware image, the *SHA-256* hashing algorithm [22] and an elliptic curve cipher based on *Curve25519* [23] are used, which are both considered modern and secure methods for software signing (see [24], [25]). The cryptographic signature for each of the two firmware images is created by the continuous integration system during build time and is provided as meta-information along with the firmware images. Therefore, the CI system must be equipped with the private key used to create the signatures. In contrast, to be able to verify the cryptographic signature the micro controller only needs to know the according public key. For the same reason as stated in Section V-B, the signature of the new firmware image can not be verified before it is written to flash.

Therefore, the calculation of the *SHA-256* checksum required for the signature check is done while the update is downloaded and written to flash. After the download has succeeded, the checksum is verified against the signature and the bootloader gets reconfigured iff the signature is validated successfully. Otherwise, the bootloader will not be reconfigured and the system will not start the invalid firmware.

VI. Implementation

Implementing *OTA* updates under the given requirements involves multiple components, which interact closely. The continuous integration system is in charge of building the firmware from source, calculating cryptographic signatures, and publishing the built firmware images. The deployment infrastructure provides resources for downloading the firmware images and triggering the update on all devices. Finally, the implementation of the update mechanism, as a part of the firmware running on the embedded device, is responsible for downloading and installing the updates.

A. Build infrastructure and automatic deployment

The CI system, which is based on *drone* [26] allows to execute commands, whenever a new version is published into the projects *Git* repository. A corresponding *drone* configuration called `.drone.yml` exists beside the source code (Figure 3). Within this configuration file, settings relevant to the build process are provided to the build environment. First, the `CONFIG=maglab` option lets the build system use an additional configuration file (`Configurion.mk.maglab`), which is stored inside the framework repository and provides environment specific information, such as the WiFi SSID. To keep secrets like the WiFi password and the private key unexposed, it is not written down in the configuration file. Instead, to include secrets into a build process while allowing to keep the configuration public, *drone* allows to encrypt these with a repository specific key. Using this method, the secrets are stored as `.drone.sec` file inside the repository from where they are injected into the build environment. Also noticeable in Figure 3 is the firmware version, which is configured to be the first 8 letters of the *Git* commit hash uniquely identifying a version of the source code. For deployment, only the master branch is considered. After a successful build, all distribution files (the firmware image and meta-information files) of all device types are copied to the repository server, from where they are served by a *HTTP 1.1* [27] server. The configuration file (`Configurion.mk.maglab`) references exactly this repository server as the source for updates.

```
build:
  image: maglab/sming
  environment:
    - CONFIG=maglab
    - WIFI_PWD=$$WIFI_PWD
    - VERSION=$${COMMIT:0:8}
  commands:
    - make clean && make
```

Figure 3. The *drone* configuration for the *ESPer* project.

Support for multiple devices of different type is implemented in both, the *ESPer* framework itself and the build system. The framework keeps control over the application lifecycle. It ensures that device unspecific code is executed at the right time and provides an API for device specific functionality. For this, a simple interface is specified by the framework, which must be implemented by each device. A single function `Device* getDevice()` must be defined exactly once in each device specific folder. To implement this interface, a static instance of `Device` is created and returned. Each `Device` is populated with device specific `Feature` instances. While the `Feature`-API leverages common run time polymorphism to share functionality between features, the initial `Device` creation uses compile time polymorphism, which reduces the need for memory management and increases performance by avoiding virtual function tables. Figure 4 shows the complete device specific code used for a simple power socket, which is mainly confined to the device type and its capabilities (e.g., the GPIO pin numbers to use).

```
constexpr const char NAME[] = "socket";
constexpr const uint16_t GPIO = 12; // General purpose I/O

Device device;
OnOffFeature<NAME, 12, false, 1> socket(&device);

Device* getDevice() { return &device; }
```

Figure 4. Device specific code for a socket driver.

The actual compilation of the source code is mainly controlled using two *Makefiles*. The first one is a helper *Makefile* built to accept a parameter for device type identifiers called `DEVICE`, and to create its whole output inside a subdirectory specific to the device type. In addition, the primary *Makefile* scans a project subdirectory and uses each directory in there as a container for device specific code. For each of these directories, the helper *Makefile* is called and the subdirectories name is used as the value of the `DEVICE` parameter. By splitting the build and recompiling the framework each time before intermixing it with the device specific code, the device type identifier can be used inside the shared framework code. While building a devices firmware, the meta-information file used during updates is also created and stored beside the firmware image. For development, each device can be build separately by using the device type identifier as *Makefile* target. In addition, the suffix `/flash` can be used to flash a specific firmware to the device.

While building the firmware images for a device, the build environment provides some constants, which are baked into the resulting firmware image. Beside the environmental configuration like the WiFi credentials, *MQTT* topics and other configurable tweaks, the current device and version identifiers are provided as compile time constants. In addition, the public key used to verify firmware signatures during updates is derived from the private key and provided as a object file, which is linked into each firmware image (Figure 5). This allows to use all the information inside the code without any overhead while being configurable during build time.

As the *ESP-01s* is only equipped with 1 MB of flash, this means that the whole memory is mapped to a contiguous address space (refer to Section V-B). Therefore, the second ROM slot can not be re-mapped to have the same start address as the first ROM slot. While the firmware is executed without any dynamic linking mechanism and the chip does not support position independent code, the addresses used in the ROM slots are dependent to the offset at which the firmware is stored. This arises the need for building two firmware images, one

for each target location. To do so, a linker script for each of the two ROM slots was created, which is used to create two variations of the same firmware, only differing in ROM placement. The two resulting firmware image files are both provided for download via *HTTP 1.1* - which one to download depends on the target ROM slot and is selected by the device during the update process. Figure 6 shows the only difference between the two linker scripts, where `${SLOT}` is replaced with the slot number according to the current build.

```
update_key_pub.bin:
    echo "$(UPDATE_KEY)" | ecdsakeygen -p | xxd -r -p > "$@"

update_key_pub.o: update_key_pub.bin
    $(OBJCOPY) -I binary $< -B xtensa -O elf32-xtensa-le $@
```

Figure 5. Creating the linker object containing the public key.

The build process will create the two firmware images, one for each ROM slot, and the meta-information file. To create the meta-information file, the current version identifier is written to the `.version` file. After the build, the signatures for both firmware images are created and attached to the file. Due to modern compilers doing link time optimization, the resulting firmware images include only code needed according the actual configuration.

```
irom0_0_seg :
  org = ( 0x40200000          // The memory mapping address
        + 0x2010              // Bootloader code and config
        + 1M / 2 * ${SLOT} ), // Offset for the ROM slot
  len = ( 1M / 2 - 0x2010  )  // Half ROM size excl. offset
```

Figure 6. Linker script to build firmware for two ROM slots.

B. The update mechanism

The update mechanism is split into four main phases: checking for updates, reprogramming the device, calculating and verifying the cryptographic signature of the updated firmware, and - assuming that the update was successful - reconfiguring the boot process to use the new firmware.

1) Checking for updates: In order to inform the IoT devices of the availability of a new firmware version, the update server provides a file for each device type containing meta-information about the latest available firmware version. The meta-information file has a simple line oriented ASCII format, which is easy to generate and efficient to parse within the limited constraints of the embedded device. It consists of the version identifier and the cryptographic signatures of both of the firmware binaries. The version identifier can be an arbitrary string as the content is not interpreted semantically but only compared to the version identifier used during build time. The other two lines in the meta-information file provide the hexadecimal representation of the cryptographic signatures, one line for each firmware binary file. These meta-information files are provided by the update server using *HTTP 1.1* [27] under the following path pattern: `${DEVICE}.version` (whereas `${DEVICE}` gets replaced by the device type name). Each device queries the update server regularly for the currently available firmware version. It uses the `UPDATER_URL` option to identify the update server. After the meta-information file has been downloaded successfully, the version identifier is extracted and compared to the version identifier of the running firmware. If the version identifiers differ, the update process is initialized. In cases where the download fails, the update server or network connection is not available, or any other error occurres, another attempt will be made automatically at the next regular interval. In addition to the interval, a special *MQTT* topic shared by all devices is subscribed on device startup: `${MQTT_REALM}/update`. Every time a message is received on this topic, a fetch attempt for the meta-information file is triggered and the process restarts. This allows faster roll-outs of updates and finer control for manual maintenance.

2) Reprogramming the device: The firmware files provided on the update server are the exact same ones as used to initially flash the chip for the according version. Using the same files for flashing and updating allows better debugging by eliminating errors related to the update process itself and eases development and initial installation. Figure 7 shows the algorithm used to determine the download address and reconfigure the bootloader. The update server provides these files in the exact same way as it provides the meta-information files, but the path pattern differs: the suffixes `.rom{0,1}` are used to provide the firmware image files for the first and second slot respectively. For installing a firmware update, the new firmware image file is downloaded using an *HTTP 1.1* `GET` request.

```
#define URL_ROM(slot) (( URL "/" DEVICE ".rom" slot ))

// Select rom slot to flash
const auto& bootconf = rboot_get_config();
if (bootconf.current_rom == 0) {
  updater.addItem(bootconf.roms[1], URL_ROM("1"));
  updater.switchToRom(1);
} else {
  updater.addItem(bootconf.roms[0], URL_ROM("0"));
  updater.switchToRom(0);
}
```

Figure 7. Configuring the updater to download the right firmware image and update the booloader accordingly.

3) Verifying the cryptographic signature: While the image is being downloaded, each chunk received in the download stream is used to update the *SHA256* hash before it is written to the flash. When the write has been finished, the next chunk is received and the process continues until all chunks have been processed. After downloading the new firmware image has been finished successfully, the calculated hash is checked against the signature of the according firmware image. Therefore, the cryptographically signed hash, which was provided in the meta-information file triggering the update, is verified against the *Curve25519* public key stored as a constant in the running firmware. Only if the checksum matches the provided signature, the firmware is considered valid and the process is continued.

4) Reconfiguring the boot process: For the bootloader, *rBoot*[28] has been choosen as it is integrated within the *Sming* framework and allows to boot to multiple ROM slots. For configuration, an *rBoot* specific structure is placed in the flash at a well-known location directly after the space reserved for the bootloader code. This structure contains, among other things, the target offsets for all known ROM slots and the number of the ROM slot to boot on next startup. To switch to the updated ROM slot after successful installation, the number ROM slot to boot on startup is changed in the configuration section and the device is restarted.

 ISBN: 978-1-61208-591-3

VII. Conclusion

In this article, we have presented a concept for building and publishing cryptographically secure *Over The Air* updates for embedded devices based on ESP8266 microcontrollers. A proof of concept implementation has been developed, which is now an essential part of the home-automation development and deployment in the *Magrathea Laboratories e.V.* hackerspace. All of the devices running the OTA-enabled firmware have undergone multiple major updates without any problems. This includes a major network configuration change and an important stability fix for the network communication stack. All devices applied the update successfully and started to work without any manual interaction required afterwards.

While the devices from various manufacturers in the hackerspace are all delivered with a pre-installed firmware, which is thought to be ready for smart home application, none of them has been provided with updates by the manufacturer so far. It is not visible to the users if the current firmware of these devices is at the latest version nor which versions are installed or how to update them.

The update infrastructure has been the crucial point for most of our members towards the framework. Enabling the developers to do updates in combination with the shared configuration and behavior provided by the framework resulted in a massive speedup when it comes to project deployment. Before that, the cost for applying changes after deployment was estimated so high, that most projects tend to delay deployment until all required and wanted features were implemented. Now, as the devices are deployed as soon as the hardware is considered stable, these devices start to provide functionality early and therefore the developers can get better feedback on the provided functionality.

The project will be continued to extend the functionality and security with features already being in development. The latest development includes further security enhancements by implementing checksum verification during startup where the hash of the firmware image is checked on each boot by the bootloader to detect tempering and defects. It also considers including the device identifier into the signature to prevent confounding of images between different device types. Last, the standby ROM slot will be updated right after each successful update to be more failsafe.

In addition, the information provided by the device about the firmware status will be enhanced to allow better control and reduce maintenance effort even more. A web interface to review the published information is currently in development.

References

[1] "Industry 4.0: A Cost and Energy efficient Micro PLC for Smart Manufacturing," Indian Journal of Science and Technology, vol. 9, no. 44, Nov. 2016.

[2] "Design of a low cost multipurpose wireless sensor network," in 2015 IEEE International Workshop on Measurements and Networking (M&N). IEEE, 2015, pp. 1–6.

[3] "ESP8266 based implementation of wireless sensor network with Linux based web-server," in 2016 Symposium on Colossal Data Analysis and Networking (CDAN). IEEE, 2016, pp. 1–5.

[4] "MQTT based home automation system using ESP8266," in 2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC). IEEE, 2016, pp. 1–5.

[5] "An IOT by information retrieval approach: Smart lights controlled using WiFi," in 2016 6th International Conference - Cloud System and Big Data Engineering (Confluence). IEEE, 2016, pp. 708–712.

[6] A. R. Beresford, "Whack-A-Mole Security: Incentivising the Production, Delivery and Installation of Security Updates," in IMPS@ ESSoS, 2016, pp. 9–10.

[7] P. Ruckebusch, E. De Poorter, C. Fortuna, and I. Moerman, "GITAR - Generic extension for Internet-of-Things Architectures enabling dynamic updates of network and application modules." Ad Hoc Networks, 2016.

[8] "Decentralized coordination of dynamic software updates in the Internet of Things," in 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT). IEEE, 2016, pp. 171–176.

[9] K. Mansour, W. Farag, and M. ElHelw, "AiroDiag: A sophisticated tool that diagnoses and updates vehicles software over air," in 2012 IEEE International Electric Vehicle Conference (IEVC), year = 2012, pages = 1–7, publisher = IEEE.

[10] D. K. Nilsson and U. E. Larson, "Secure Firmware Updates over the Air in Intelligent Vehicles," in ICC 2008 - 2008 IEEE International Conference on Communications Workshops. IEEE, 2008, pp. 380–384.

[11] S. Gore, S. Kadam, S. Mallayanmath, and S. Jadhav, "Review on Programming ESP8266 with Over the Air Programming Capability," International Journal of Engineering Science, vol. 3951, 2016.

[12] Magrathea Laboratories e.V., "Magrathea Laboratories - Creating new Worlds," URL: https://maglab.space/, [accessed: 2017.05.22].

[13] Home Assistant, "Awaken your home," http://home-assistant.io/, [accessed: 2017.05.22].

[14] ESPRESSIF, "ESP8266 Overview," URL: http://www.espressif.com/en/products/hardware/esp8266ex/overview, [accessed: 2017.05.22].

[15] SparkFun, "WiFi Module - ESP8266," URL: https://www.sparkfun.com/products/13678, [accessed: 2017.05.22].

[16] ITEAD, "Sonoff Smart-home," URL: https://www.itead.cc/smart-home.html, [accessed: 2017.05.22].

[17] Sming, "Sming - Open Source framework for high efficiency native ESP8266 development," URL: http://sminghub.github.io/Sming/about/, [accessed: 2017.05.22].

[18] OASIS Standard Incorporating, "MQTT Version 3.1.1 Plus Errata 01," URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html, [accessed: 2017.05.22].

[19] git, "git - a free and open source distributed version control system," URL: https://git-scm.com, [accessed: 2017.05.22].

[20] The IEEE and The Open Group, "The Open Group Base Specifications Issue 6 - make - maintain, update, and regenerate groups of programs," URL: http://pubs.opengroup.org/onlinepubs/009695399/utilities/make.html, [accessed: 2017.05.22].

[21] E. Community, "ESP8266 Memory Map," URL: http://www.esp8266.com/wiki/doku.php?id=esp8266_memory_map, [accessed: 2017.05.22].

[22] D. Eastlake and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," Internet Requests for Comments, RFC Editor, RFC 6234, May 2011, http://www.rfc-editor.org/rfc/rfc6234.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6234.txt

[23] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in International Workshop on Public Key Cryptography. Springer, 2006, pp. 207–228.

[24] E. Barker and Q. Dang, "NIST Special Publication 800–57 Part 1, Revision 4," 2016.

[25] F. O. for Information Security, "Cryptographic Mechanisms: Recommendations and Key Lengths," Online, Federal Office for Information Security, BSI Technical Guideline BSI TR-02102-1, February 2017, URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf, [accessed: 2017.05.22].

[26] Drone, "Drone is a Continuous Delivery platform built on Docker, written in Go," URL: https://github.com/drone/drone, [accessed: 2017.05.22].

[27] The Internet Society, "Hypertext Transfer Protocol – HTTP/1.1," URL: https://www.w3.org/Protocols/rfc2616/rfc2616.html, [accessed: 2017.05.22].

[28] R. A. Burton, "An open source bootloader for the ESP8266," URL: https://github.com/raburton/rboot, [accessed: 2017.05.22].

 ISBN: 978-1-61208-591-3