Müller, Neumair, Reiser, Dreo Rodosek (Hrsg.): 11. DFN-Forum Kommunikationstechnologien,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 69

Realisierung von sicheren Over-the-Air Updates für ESP8266-basierte IoT-Endgeräte

Dustin Frisch[1] Sven Reißmann[2] Christian Pape [1] Sebastian Rieger[1]

Abstract: Nicht zuletzt durch Trends wie Smart Home und Industrie 4.0 hat die Zahl der Endgeräte im Bereich Internet of Things (IoT) in den vergangenen Jahren stark zugenommen. Häufig werden diese Endgeräte als kostengünstige kleine eingebettete Systeme in großen Stückzahlen ausgerollt und über viele Jahre verwendet. Sie stellen nicht nur bedingt durch ihre knappen Ressourcen und eingeschränkten Schutzmechanismen ein Datenschutzrisiko für ihre Betreiber dar, sondern bilden aufgrund ihrer großen Verteilung und Angriffsfläche auch ein Sicherheitsrisiko für das gesamte Internet. Um diesem Risiko entgegenzuwirken, müssen Sicherheitsupdates für IoT-Endgeräte regelmäßig und zeitnah verteilt werden. Dies verdeutlichen auch die in den letzten Jahren aufgetretenen Angriffe, die auf IoT-Endgeräten aufgesetzt haben (z.B. Mirai Botnet). Das vorliegende Paper beschreibt ein nachhaltiges und stabiles Verfahren für die Bereitstellung kryptographisch gesicherter Over-the-Air Firmware-Updates für eingebettete Systeme basierend auf dem verbreiteten ESP8266 Mikrocontroller. Im Gegensatz zu anderen Over-the-Air-Verfahren werden der unterbrechungsfreie Betrieb des Mikrocontrollers und fehlertolerante Updates realisiert. Da der ESP8266 aufgrund seiner knappen Ressourcen keine vollständige HTTPS- bzw. TLS-Unterstützung bietet, wurde eine separate kryptographische Überprüfung bereitgestellter Updates implementiert. Das vorgestellte Verfahren umfasst den Build-Prozess des Updates, dessen automatische digitale Signatur sowie die Verteilung und Installation auf den Endgeräten. Darüber hinaus wird die Verwendung des Verfahrens zur Abwehr von auf IoT-Endgeräten basierenden Angriffen aufgezeigt.

Keywords: Internet of Things; Secure Remote Firmware Updates; Over-the-Air Updates; ESP8266

1 Einleitung

Das Internet of Things (IoT) hat zu einem großen Wachstum von sogenannten Smart Devices geführt, die z.B. Sensoren und Aktoren über das Internet miteinander verbinden. Bestehende Geräte wie z.B. Türschlösser, Lampen, Waschmaschinen etc. werden hierbei um smarte Funktionen erweitert, um sie z.B. aus der Ferne zu steuern und zu überwachen. Um diese smarten Funktionen zu implementieren, werden kleine eingebettete Systeme in die Geräte eingebaut, die z.B. die Anbindung an ein drahtloses Netz ermöglichen. Implementiert werden die Funktionen als Software, bzw. als Firmware, auf den eingebetteten Systemen. Die Firmware dient dabei zum Einen der Anbindung spezifischer Hardware (vgl. Sensoren und Aktoren) und bietet zum Anderen allgemeine Funktionen, vergleichbar der Aufgabe eines

[1] Hochschule Fulda, Angewandte Informatik, [vorname.nachname]@informatik.hs-fulda.de
[2] Hochschule Fulda, Rechenzentrum, sven.reissmann@rz.hs-fulda.de

cc by sa

Betriebssystems, unabhängig vom konkreten Einsatzgebiet (z.B. Anbindung an WLAN-Netze etc.). Anforderungen und Funktionen von eingebetteten Systemen werden häufig als unveränderlich angenommen. Allerdings zeigt sich beim Einsatz der Systeme in der Realität, dass sich allein durch Anforderungen der Umgebung in der sie betrieben werden, häufig doch Veränderungen ergeben, an die die Systeme angepasst werden müssen. Beispiele hierfür sind Veränderungen oder Erweiterung des Verhaltens der Systeme, Anpassungen der Netzanbindung (vgl. Reaktion auf neue Angriffe etc.) oder Fehlerbehebung bzw. insb. Sicherheitsupdates im Laufe des Betriebs. In den meisten Fällen können diese Anpassungen allein durch die Änderung der Firmware erfolgen, während die Hardware unverändert bleibt. Um die Firmware zu aktualisieren, müssen die Systeme eine geeignete Schnittstelle bieten, die in der Regel auch die Konfiguration des Systems sowie Informationen und Checks zur aktuellen Firmware verwaltet. In aller Regel erfordern die Schnittstellen für das Firmware-Update einen direkten physischen Zugriff auf das System. Insbesondere bei einer großen Anzahl und räumlichen Verteilung von Endgeräten sind Firmware-Updates, die einen direkten Zugriff auf das System erfordern, jedoch kaum realisierbar. Abhilfe bieten Schnittstellen, die ein *Over-the-Air (OTA)* Update aus der Ferne ermöglichen, solange das Endgerät über ein Netz erreicht werden kann. Das Update kann in diesem Fall ohne direkten Zugriff über die reguläre Netzwertschnittstelle erfolgen, ohne eine spezielle Schnittstelle zu erfordern. *OTA*-Updates ermöglichen ein automatisiertes Ausrollen von neuen Firmware-Versionen und Patches auf eine große Anzahl verteilter Endgeräte. Die Automatisierung kann dabei im Sinne einer Continuous Delivery Tests und Prozesse für die fehlertolerante Auslieferung der Firmware beinhalten. Hierbei können Updates z.B. zunächst automatisiert auf Testgeräten eingespielt und überprüft werden. Sicherheitsupdates können priorisiert im laufenden Betrieb ausgerollt werden, während Funktionsupdates verzögert verteilt werden, z.B. sobald das Endgerät zeitweise nicht verwendet wird. Über das Netz kann zusätzlich ein Monitoring des Endgeräts und der Firmware-Updates erfolgen, um sicherzustellen, dass alle Updates erfolgreich waren und sich das Endgerät im gewünschten Zustand befindet.

Die verbleibenden Abschnitte des Papers gliedern sich wie folgt. Abschnitt 2 nennt verwandte Arbeiten. In Abschnitt 3 werden die Anforderungen an das im nachfolgenden Abschnitt 4 erstellte Konzept für einen sicheren und robusten *OTA*-Update-Mechanismus definiert. Eine Referenzimplementierung folgt im Abschnitt 5. Abschließend zieht Abschnitt 6 ein Fazit und gibt einen Ausblick auf weitere Arbeiten.

2 Verwandte Arbeiten

Drahtlose Netze für Sensoren und Aktoren sind eine essentielle Grundlage für die Realisierung von Industrie 4.0 Infrastrukturen und modere Fertigungs- und Lieferprozesse. Günstige Programmable Logic Controller (PLC) und Cloud Computing ermöglichen und treiben diese neuen Paradigmen im Produktionsbereich gleichermaßen [NS16]. In [Di15] wird ein multifunktionales, kostengünstiges und drahtloses Sensornetz unter Verwendung des *ESP8266* PLCs vorgestellt. Die Verwendung eines *ESP8266* PLCs in Kombination mit

einem Raspberry Pi als Basisstation wird in [Th16] gezeigt. Der Artikel [KS16] präsentiert eine Heimautomatisierungslösung basierend auf einer *MQTT* Message Queue mit *ESP8266*-basierenden Sensoren und Aktoren. Die Kontrolle von smarten Lampen mit PLCs wird in [WKM16] zusammengefasst. Firmware-Update-Mechanismen werden in diesen Veröffentlichungen leider nicht adressiert. Die Bedeutung regelmäßiger Sicherheitsupdates für aktuelle IT-Infrastrukturen wurde in [Be16] zusammengestellt. Ein Ansatz für dezentrale Software Updates in Contiki-basierten IoT-Umgebungen wurde in [Ru16] vorgestellt. In [We16] werden Software Updates für Java-basierte Endgeräte präsentiert. Beide Lösungen sind für kleine Mikrocontroller mit geringen Ressourcen nicht anwendbar. [MFE12] beschreibt ein Diagnose- und Update-System für Embedded Software von Electronic Control Units in Fahrzeugen. Die Verwendung sicherer Firmware-Updates in der Automobilindustrie wird in [NL08] behandelt. In [Go16] wird ein Konzept für *Over-the-Air* Updates für *ESP8266* PLCs beschrieben. Die Updates sind allerdings nicht fehlertolerant im laufenden Betrieb möglich und erfordern ein separates Gateway.

3 Anforderungen

Sobald ein neues Firmware-Release vorliegt, soll das Update automatisch und ohne manuelle Interaktion auf die Endgeräte ausgerollt werden. Endgeräte sollen für sie passende neue Firmware-Versionen erkennen, herunterladen, automatisch installieren und, sofern bei der Installation kein Fehler auftritt, die neue Firmware durch einen Neustart anwenden. Um einen minimalen Wartungsaufwand zu erzielen, soll der Update-Prozess möglichst robust und fehlertolerant sein. Sofern während einem Update ein Fehler auftritt, soll das Endgerät weiterhin fehlerfrei funktionieren. Firmware-Updates müssen über die WLAN-Verbindung während des regulären Betrieb möglich sein ohne die Funktion des Geräts zu beeinträchtigen. Der Update-Prozess muss über nicht vertrauenswürdige WLAN-Netze und das Internet sicher möglich sein. Um zu verhindern, dass Angreifer manipulierte Firmware-Versionen einspielen können, soll eine digitale Signatur der Updates realisiert und überprüft werden. Der Update-Prozess soll nur dann angestoßen werden, wenn neue Firmware-Versionen vorliegen, und möglichst schnell ablaufen, um den unterbrechungsfreien Betrieb des Endgeräts zu sichern. Zur Erleichterung der Verwaltung und Überwachung von Endgeräten sollen für den Update-Prozess relevante Informationen (z.B. installierte Firmware-Version) von den Endgeräten abrufbar sein. Bei einem Update muss sichergestellt werden, dass nur die für die jeweilige Hardware passende Firmware verwendet wird.

4 Konzept für die Implementierung von *OTA*-Updates

Um die im vorherigen Abschnitt genannten Anforderungen an *OTA*-Updates zu erfüllen, wurde zunächst eine IoT-Infrastruktur realisiert, die den Build-Prozess für die Firmware, das zugehörige Repository sowie das WLAN-Netz mit den IoT-Endgeräten und deren Controller umfasst. Für die in diesem Paper verwendete Implementierung wurde bewusst auf

leichtgewichtige, gängige Software-Projekte gesetzt, um die Austauschbarkeit individueller Komponenten zu erleichtern. Abbildung 1 zeigt die verwendete IoT-Umgebung.

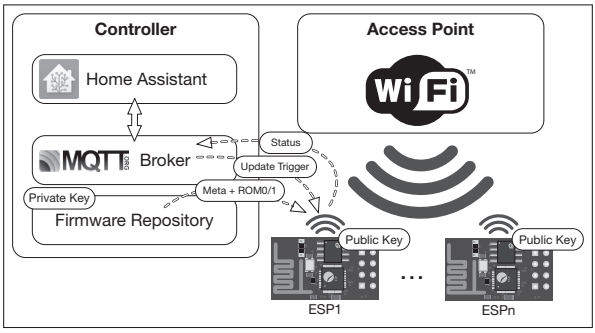

Abb. 1: Verwendete Netztopologie und Systemarchitektur.

Um die zentrale Verwaltung und Überwachung von Endgeräten zu ermöglichen, sendet jedes Endgerät Statusinformationen an ein vordefiniertes *MQTT* Topic, sobald es mit dem Netzwerk verbunden ist. Neben Informationen zum Typ des Endgeräts sowie Chip- und Flash-ID werden Details zum Bootloader, SDK und der derzeitigen Firmware-Version sowie relevante Angaben zur Bootloader-Konfiguration, wie z.B. der von der aktuell laufenden Firmware verwendete ROM-Bereich sowie der vom Boot-Prozess standardmäßig verwendete ROM-Bereich übermittelt. Dadurch können Administratoren Endgeräte mit veralteter Firmware finden, z.B. um fehlende oder fehlgeschlagene Updates zu erkennen.

4.1 Framework und Build-Infrastruktur

Das Framework umfasst ein Build-System, welches die Konfiguration von Basisparametern für alle Endgeräte erlaubt. Diese umfassen u.a. WLAN-Zugriffsparameter, *MQTT* Verbindungseinstellungen und URLs für den Bezug von Updates. Durch die Wiederverwendung von gleichen Code-Fragmenten wird sichergestellt, dass alle Endgeräte das gleiche Verhalten z.B. in Bezug auf die Übermittlung ihres Status oder die Interaktion mit dem Heimautomatisierungs-Controller aufweisen. Dies erleichtert die Konfiguration und erlaubt die Sammlung von Informationen zu den Endgeräten an einer zentralen Stelle.

4.2 Einrichtung des Endgeräts und Flash Layout

Auf der *ESP8266* MCU basierende Mikrocontroller-Boards haben meist das gleiche Layout: die MCU ist verbunden mit einem Flash-Speicher, der den Bootloader, die Firmware und andere Anwendungsdaten speichert. Das Memory Mapping der MCU ermöglicht nur jeweils eine 1 MB Memory Page aus dem Flash abzubilden [ES]. Zusätzlich muss der Zugriff auf die Blöcke auf die Blockgrenzen (je 1 MB) ausgerichtet werden. Da die herunterzuladende

Firmware größer als der freie Memory Heap Space sein kann, müssen die empfangenen Daten direkt in den Flash-Speicher geschrieben werden. Dabei kann die Ausführung von Code aus den gleichzeitig überschriebenen Memory Mapped Flash Pages zu unerwartetem Verhalten führen, da der Schreibvorgang den Code während der Ausführung verändert. Um dieses Problem zu verhindern wurde in dem Projekt der Flash-Speicher in zwei Hälften geteilt, wobei zwei ROM Slots entstehen, die unterschiedliche Firmware-Versionen beinhalten können (vgl. Abbildung 2). Eine davon wird z.B. aktuell ausgeführt, die andere kann z.B. durch den Download eines Firmware-Updates überschrieben werden. Neben den zwei Firmware ROM Slots bietet der Flash-Speicher Platz für den Bootloader und dessen Konfiguration. Für das Alignment und ein vereinfachtes Debugging wurde der zweite ROM Slot um die Größe des Bootloaders verschoben (Padding). Die 8192 Byte große Lücke kann für das persistente Speichern von Anwendungsdaten über Firmware-Updates hinweg genutzt werden. Der zusätzliche "standby" ROM Slot dient als Sicherungsmechanismus. Schlägt ein Update fehl oder wird unterbrochen, bleibt die vorherige Firmware-Version in dem "aktiven" Slot intakt und kann ohne Ausfall des Endgeräts weiter verwendet werden.

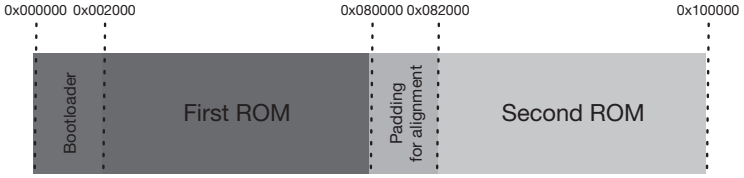

Abb. 2: Layout des Flash-Speichers zur Verwendung von zwei separaten ROM Slots.

4.3 Verschlüsselung der Firmware-Updates

Um Manipulationen und Fehler bei der Übertragung der Firmware auf das Endgerät zu verhindern, berechnet der Update-Prozess einen Hash der übermittelten Firmware und prüft diesen anhand der mitgesendeten digitalen Signatur. Hierfür wird der Hash-Algorithmus *SHA-256* [EH11] und ein Elliptic Curve Cipher basierend auf *Curve25519* [Be06] verwendet, wie u.a. derzeitig in [BD16; Fe17] als sicheres Verfahren für die Signatur von Software empfohlen. Die digitale Signatur wird während des Build-Prozesses mit Hilfe des privaten Schlüssels erzeugt und als Metainformation zum Update bereitgestellt. Als Gegenstück dazu nutzen die Mikrocontroller den zugehörigen öffentlichen Schlüssel für die Überprüfung der digitalen Signatur nach dem Empfang des Updates. Wie bereits im Abschnitt 4.2 erläutert, muss die neue Firmware direkt auf den Flash-Speicher geschrieben werden. Entsprechend wird der *SHA-256* Hash während des Downloads und dem Schreibvorgang auf den Flash-Speicher ermittelt. Nachdem der Download erfolgreich beendet wurde, wird der Hash-Wert anhand der digitalen Signatur verifiziert. Sofern die Überprüfung erfolgreich ist, wird die Bootloader-Konfiguration geändert und die neue Firmware aktiviert. Andernfalls bleibt die alte Firmware des Endgeräts aktiv und der Neustart entfällt.

5 Implementierung

Für *OTA*-Updates ist das Zusammenspiel verschiedener Komponenten bzw. Systeme nötig. Dabei übersetzt das Build-System den Quelltext und erzeugt die Firmware-Dateien inklusive zugehöriger digitaler Signaturen. Die Deployment-Infrastruktur stellt die Firmware-Dateien bereit und löst die Aktualisierung der Geräte aus. Die Implementierung des Update-Mechanismus als Teil der installierten Firmware des eingebetteten Systems ist selber verantwortlich für den Lade- und Installationsprozess der aktualisierten Firmware.

5.1 Build und Deployment

Sowohl das in diesem Paper vorgestellte ESPer Framework als auch das Build-System als solches unterstützen die Erzeugung von Firmware für unterschiedliche Geräteklassen. Dabei kontrolliert das Framework den Lebenszyklus der Firmware. Zu diesem Zweck definiert das Framework eine simple Schnittstelle, welche durch alle Geräteklassen implementiert werden muss. Dabei muss eine Instanz von *Device* erzeugt und zurückgegeben werden, welches selber wiederum Instanzen des Typs *Feature* enthält. Während die *Feature*-API es erlaubt Funktionalität durch Polymorphismus zur Laufzeit zu vereinheitlichen, wird bei der *Device*-Erzeugung Polymorphismus zur Übersetzungszeitpunkt eingesetzt um sowohl die Speicherverwaltung zu vereinfachen als auch virtuelle Funktionstabellen an dieser Stelle zu vermeiden. Abbildung 3 zeigt den kompletten gerätespezifischen Quellcode für eine simple schaltbare Steckdose, welcher sich auf die Definition des Typs und dessen Eigenschaften beschränkt (bspw. die zu nutzenden GPIO-Pins).

```
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; }
```

Abb. 3: Gerätespezifischer Quellcode für eine schaltbare Steckdose.

Während der Übersetzung und Erzeugung der Firmware-Dateien werden auch deren Metainformationen als Datei in das jeweilige Verzeichnis gespeichert. Zusätzlich werden Konstanten aus der Build-Umgebung direkt in die resultierenden Firmware-Dateien geschrieben. Neben den WLAN-Zugangsdaten, den *MQTT*-Topics und weiteren konfigurierbaren Parametern stehen zusätzlich der Gerätetyp und die Firmware-Version als Konstanten zur Verfügung. Darüber hinaus wird auch der öffentliche Schlüssel zur Verifizierung der Firmware-Signaturen aus dem privaten Schlüssel generiert und als Objekt-Datei gegen jedes Firmware-Image gelinkt (Abbildung 4). Dies erlaubt die Nutzung aller Informationen im Quelltext ohne Einschränkungen obwohl diese erst zur Übersetzungszeit konfigurierbar bzw. bekannt sind. Der *ESP-01s* ist lediglich mit 1 MB Flash-Speicher ausgestattet, wobei dieser als einzelner kompletter Adressbereich zur Verfügung steht (siehe Abschnitt 4.2). Daher

kann der zweite ROM Slot nicht die gleiche Start-Adresse wie der erste ROM Slot nutzen. Da die Firmware ohne einen dynamischen Linking-Mechanismus ausgeführt wird und der ESP keinen positionsunabhängigen Code unterstützt, ist es nötig, dass die Adressierung je nach Offset der Firmware im ROM angepasst wird. Aus diesem Grund werden durch zwei Linker-Skripte die zwei Ausprägungen der Firmware erstellt, je nach Zielposition innerhalb des Speichers. Die zwei resultierenden Firmware-Images werden beide via *HTTP 1.1* zur Verfügung gestellt. Welches schließlich heruntergeladen und installiert wird hängt vom Zielslot ab. Abbildung 5 zeigt die Unterschiede der zwei Linker-Skripte, wobei *${SLOT}* die Slotnummer für den aktuellen Übersetzungsprozess enthält.

```
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 $@
```

Abb. 4: Erzeugung der Objekt-Datei für den öffentlichen Schlüssel.

```
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
```

Abb. 5: Linker Skript zur Erstellung der Firmware für die zwei unterschiedlichen ROM Slots.

Der Erstellungsprozess erzeugt neben den zwei Firmware-Images auch die dazugehörigen Dateien mit den Metainformationen. Für deren Erzeugung wird die aktuelle Version in eine Datei *.version* geschrieben. Nach der Fertigstellung werden die Signaturen der beiden Firmware-Dateien erzeugt und den Dateien entsprechend angehängt. Nach der erfolgreichen Erzeugung der Firmware und der zugehörigen Dateien (Metainformationen) für alle Geräteklassen, werden diese auf den Repository-Server kopiert, wo diese mittels *HTTP 1.1* zur Verfügung gestellt werden. Dieser wird auch in der Konfigurationsdatei des Projekts angegeben und gilt für die Geräte als Quelle für Aktualisierungen.

5.2 Update-Prozess

Der Aktualisierungsvorgang besteht aus vier Phasen: Prüfung auf Aktualisierungen, Reprogrammierung des Geräts, Berechnung und Verifizierung der digitalen Signaturen der zu installierenden Firmware und schließlich - im Falle der erfolgreichen Aktualisierung - der Neukonfiguration des Boot-Prozesses zur Nutzung der neue Firmware. Um die IoT-Geräte über die Verfügbarkeit neuer Firmware-Versionen zu informieren hält der Update-Server für jede Geräteklasse eine Datei mit den Metainformationen zur letzten verfügbaren Firmware-Version. Diese Metainformationen enthalten sowohl die Versionsnummer als auch die digitalen Signaturen beider Firmware-Dateien. Diese Informationen werden durch den Update-Server mit Hilfe von *HTTP 1.1* als *${DEVICE}.version* zur Verfügung gestellt

(wobei *${DEVICE}* jeweils durch die Geräteklasse substituiert wird). Jedes Gerät fragt regelmäßig den durch die *UPDATER_URL* spezifizierten Update-Server nach der verfügbaren Firmware-Version. Dabei werden die Metainformationen heruntergeladen und die Versionsnummer mit der aktuell installierten Version verglichen. Falls sich diese unterscheiden wird der eigentliche Aktualisierungsprozess initiiert. Sofern Fehler beim Download auftreten, wird der Versuch beim nächsten Überprüfungsintervall wiederholt. Abbildung 6 zeigt die Bestimmung der Download-Adresse und die Neukonfiguration des Bootloaders. Dabei stellt der Update-Server die Firmware-Dateien analog zu den Metainformationen zur Verfügung. Durch Ergänzung des Pfades mit *.rom{0,1}* kann somit auf das Firmware-Image für den ersten bzw. zweiten ROM-Slot zugegriffen werden. Die gewählte Datei wird dann entsprechend über *HTTP 1.1 GET* vom Update-Server heruntergeladen.

```
#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); }
```

Abb. 6: Download-Adresse und Neukonfiguration des Bootloaders je nach verwendetem ROM-Slot.

Die Firmware wird in Teilstücken vom Update-Server heruntergeladen. Dabei wird zunächst die *SHA256* Checksumme aktualisiert bevor das Teilstück in den Flash-Speicher geschrieben wird. Nach dem erfolgreichen Schreiben wird mit dem nächsten Teilstück fortgefahren. Bei erfolgreichem Abschluss des Vorgangs wird der resultierende Hashwert gegen die Signatur des Firmware-Images geprüft. Dazu wird der kryptografisch signierte Hashwert aus den Metainformationen gegen den *Curve25519* öffentlichen Schlüssel der installierten Firmware geprüft. Nur wenn die Checksummen mit der gegebenen Signatur übereinstimmen wird die Firmware als valide angenommen und der Aktualisierungsprozess fortgesetzt. Als Bootloader wird *rBoot* [Bu] verwendet, da dieser im *Sming* Framework integriert ist und unterschiedliche ROM-Slots booten kann. Zur Konfiguration muss eine *rBoot*-spezifische Struktur an eine definierte Stelle im Flash geschrieben werden. Diese Struktur enthält die Ziel-Offsets für alle bekannten ROM-Slots und die Nummer des ROM-Slots der während des Bootvorgangs verwendet werden soll. Um diesen nach einer erfolgreichen Aktualisierung zu wechseln wird die Struktur angepasst und ein Neustart des Geräts initiiert.

Falls die die kryptografische Signatur nicht validiert werden kann wird die aktuelle Konfiguration beibehalten und der Neustart entfällt. Ein weiterer Update-Versuch wird dann nach Ablauf des eingestellten Intervalls oder durch erneute Signalisierung eines verfügbaren Updates unternommen.

6 Fazit und Ausblick

In diesem Paper wurde ein Konzept für die Erzeugung und Verteilung von kryptographisch gesicherten *Over-the-Air*-Updates für IoT-Endgeräte basierend auf dem ESP8266 Mikrocontroller vorgestellt. Die entwickelte Proof of Concept Implementierung ist essentieller Bestandteil des Home-Automation Development und Deployment im *Magrathea Laboratories e.V.* Hackerspace [Ma]. Alle Endgeräte, die in dieser Umgebung eine *OTA*-unterstützende Firmware verwenden wurden mit der hier vorgestellten Lösung mehrfach erfolgreich und ohne Probleme im laufenden Betrieb aktualisiert, ohne dass eine manuell Intervention erforderlich war. Während des Updates war die Funktion der Geräte nicht eingeschränkt und sie konnten bis zum Zeitpunkt des Updates wie gewohnt verwendet werden. Hierbei wurden auch Änderungen an der Netzkonfiguration und wichtige Stabilitätsupdates am Netzwerk-Stack durchgeführt. Durch die Entwicklung des in diesem Paper vorgestellten Frameworks für einen automatisierten Build- und Update-Prozess wird dieses Problem adressiert und eine einheitliche Umgebung für die sichere Verteilung von Firmware-Updates für IoT-Endgeräte bereitgestellt. Dadurch konnte der Entwicklungsprozess vereinfacht werden, Änderungen im Code schneller in abhängige Module und Funktionen übernommen werden und eine zeitnahe Verteilung auf die Endgeräte im Sinne einer Continuous Delivery erzielt werden. Zukünftig soll die Funktionalität und Sicherheit des Frameworks zusätzlich erweitert werden. Die aktuelle Entwicklung umfasst weitergehende Sicherheitsmechanismen wie die Verifikation der Integrität der Firmware während des Boot-Vorgangs, um Manipulationen oder defekte Firmware-Images zu erkennen. Darüber hinaus wird die Aufnahme des Endgerätetyps in die digitale Signatur in Betracht gezogen, um Fehlzuordnungen der Firmware zu nicht unterstützten Endgeräten zu verhindern. Ebenfalls ist eine Übernahme der Firmware in den "standby" ROM-Slot nach jedem erfolgreichen Update geplant, um die Fehlertoleranz weiter zu steigern. Zusätzlich ist die Erweiterung der Statusinformationen und die Entwicklung einer Web-Anwendung für das Monitoring der IoT-Infrastruktur in der Umsetzung.

Literatur

[BD16] Barker, E.; Dang, Q.: NIST Special Publication 800–57 Part 1, Rev. 4, 2016.

[Be06] Bernstein, D. J.: Curve25519: new Diffie-Hellman speed records. In: International Workshop on Public Key Cryptography. Springer, S. 207–228, 2006.

[Be16] Beresford, A. R.: Whack-A-Mole Security: Incentivising the Production, Delivery and Installation of Security Updates. In: IMPS@ESSoS. S. 9–10, 2016.

[Bu] Burton, R. A.: An open source bootloader for the ESP8266, `https://github.com/raburton/rboot`, [abgerufen am: 2018-04-16].

[Di15] Di Nisio, A.; Di Noia, T.; Carducci, C. G. C.; Spadavecchia, M.: Design of a low cost multipurpose wireless sensor network. In: 2015 IEEE International Workshop on Measurements and Networking (M&N). IEEE, S. 1–6, 2015.

[EH11] Eastlake, D.; Hansen, T.: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF), RFC 6234, http://www.rfc-editor.org/rfc/rfc6234.txt, [abgerufen am: 2018-04-16], RFC Editor, Mai 2011.

[ES] ESP8266 Community: ESP8266 Memory Map, http://www.esp8266.com/wiki/doku.php?id=esp8266_memory_map, [abgerufen am: 2018-04-16].

[Fe17] Federal Office for Information Security: Cryptographic Mechanisms: Recommendations and Key Lengths, BSI – Technical Guideline BSI TR-02102-1, https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf, [abgerufen am: 2018-04-16], Federal Office for Information Security, Feb. 2017.

[Go16] Gore, S.; Kadam, S.; Mallayanmath, S.; Jadhav, S.: Review on Programming ESP8266 with Over the Air Programming Capability. International Journal of Engineering Science 3951/, 2016.

[KS16] Kodali, R. K.; Soratkal, S.: MQTT based home automation system using ESP8266. In: 2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC). IEEE, S. 1–5, 2016.

[Ma] Magrathea Laboratories e.V.: Magrathea Laboratories - Creating new Worlds, https://maglab.space/, [abgerufen am: 2018-04-16].

[MFE12] Mansour, K.; Farag, W.; ElHelw, M.: AiroDiag: A sophisticated tool that diagnoses and updates vehicles software over air. In: 2012 IEEE International Electric Vehicle Conference (IEVC). IEEE, S. 1–7, 2012.

[NL08] Nilsson, D. K.; Larson, U. E.: Secure Firmware Updates over the Air in Intelligent Vehicles. In: ICC 2008 - 2008 IEEE International Conference on Communications Workshops. IEEE, S. 380–384, 2008.

[NS16] Nigappa, K.; Selvakumar, J.: Industry 4.0: A Cost and Energy efficient Micro PLC for Smart Manufacturing. Indian Journal of Science and Tech. 9/44, 2016.

[Ru16] Ruckebusch, P.; De Poorter, E.; Fortuna, C.; Moerman, I.: GITAR - Generic extension for Internet-of-Things Architectures enabling dynamic updates of network and application modules. Ad Hoc Networks/, 2016.

[Th16] Thaker, T.: ESP8266 based implementation of wireless sensor network with Linux based web-server. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN). IEEE, S. 1–5, 2016.

[We16] Weisbach, M.; Taing, N.; Wutzler, M.; Springer, T.; Schill, A.; Clarke, S.: Decentralized coordination of dynamic software updates in the Internet of Things. In: 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT). IEEE, S. 171–176, 2016.

[WKM16] Walia, N. K.; Kalra, P.; Mehrotra, D.: An IOT by information retrieval approach: Smart lights controlled using WiFi. In: 6th International Conference - Cloud System and Big Data Engineering (Confluence). IEEE, S. 708–712, 2016.