Paper published at the 39th IEEE International Conference on Distributed Computing Systems (ICDCS) in July 2019

UpKit: An Open-Source, Portable, and Lightweight Update Framework for Constrained IoT Devices

Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer
Institute of Technical Informatics, Graz University of Technology, Austria
E-mail: antonio@langiu.com; cboano@tugraz.at; markus.schuss@tugraz.at; roemer@tugraz.at

***Abstract*—Updating the software running on constrained IoT devices such as low-power sensors and actuators in a secure and efficient way is an open problem. The limited computational, memory, and storage capabilities of these devices, together with their small energy budget, indeed, restrict the number of features that can be embedded into an update system and make it also difficult to build a generic and compact solution. As a result, existing update systems for constrained IoT devices are often not portable, do not perform a proper verification of the downloaded firmware, or focus only on a single phase of the update process, which exposes them to security threats and calls for new solutions. In this paper we present UpKit, a portable and lightweight software update framework for constrained IoT devices encompassing all phases of the update process: from the generation and signature of a new firmware, to the transmission of the latter to an IoT device, its verification and installation. UpKit employs a novel update architecture that is agnostic to how new firmware images are distributed and that introduces a double-signature process to guarantee the freshness of a new firmware. This, together with an additional verification step, allows also to reject invalid software at an early stage and to prevent an unnecessary reboot of the device. We keep UpKit's design modular and provide an open-source implementation for several operating systems, off-the-shelf hardware platforms, as well as cryptographic libraries. We further include support for differential updates and flexible memory slots, which allows to significantly increase the efficiency of the update process. An experimental evaluation shows that UpKit can be used to efficiently update highly-constrained IoT devices, and that it has a comparable memory footprint to state-of-the-art solutions, despite the introduction of several additional features.**

I. Introduction

The Internet of Things (IoT) is becoming an integral part of our daily lives, with billions of networked smart objects empowering the development of smart buildings, grids, and cities, as well as attractive applications in the area of connected health, precision agriculture, and cyber-manufacturing [1].

Smart objects are often unsupervised and operate in harsh environmental conditions for several years [2]. During this extended period of time, the software (SW) running on these devices needs to be regularly updated in order to add new features, fix bugs, and resolve known security vulnerabilities.

The inability of doing so may: (i) result in reduced performance, (ii) negatively affect customer satisfaction, and (iii) expose the device to attacks compromising the safety and privacy of the involved users [3], [4]. Even worse, compromised IoT devices may be exploited to perform massive distributed denial of service attacks able to affect critical infrastructures of the Internet, as recently shown by Mirai and other IoT botnets [5].

Therefore, in order to fix bugs, patch vulnerabilities, improve performance, and extend functionality, it is necessary to *embed some software update capabilities* in the firmware running on each IoT device [6], [7]. Such capabilities include, among others, the (over-the-air) download of the update image, the verification of its integrity and authenticity, as well as its installation [8]. All these steps should be performed while minimizing the downtime of a device and its services.

Updating constrained IoT devices. Updating the software running on constrained IoT devices, e.g., low-power sensors and actuators [9], is still an open research problem [6], [10]. These devices are typically severely limited in terms of network bandwidth, as well as computational, memory, and storage capabilities, which restricts the number of features that can be embedded into an update system [11]. The availability of only 100 to 250 kB of ROM and 10 to 50 kB of RAM [12], indeed, makes it impossible to reuse or even adapt solutions developed to update powerful gateway nodes and high-end IoT devices embedding a user interface [13]–[18], as these require sufficient memory and computational power to run Linux and its derived distributions. At the same time, low-power IoT devices are typically battery-powered and their operation needs to be highly efficient, so to preserve the limited energy budget.

Therefore, SW update solutions for constrained IoT devices need to have a small memory footprint and should minimize energy expenditure, for example, by using differential updates in order to reduce the data transferred over the network and the actual update time [19]. Furthermore, constrained IoT devices are largely heterogeneous in nature (i.e., there is a plethora of different hardware platforms) and make use of different lightweight operating systems (OS) [20], optimized networking and application-layer protocols [21], as well as cryptographic libraries [22]–[24]. This heterogeneity calls for a *lightweight* update solution that is *portable* across different hardware platforms and operating systems, as well as *efficient* and *generic* (i.e., independent of the employed protocols or libraries). Unfortunately, such a solution does still not exist, and existing approaches exhibit a large number of drawbacks.

Limitations of existing approaches. To cope with the aforementioned constraints, existing update solutions for constrained IoT devices often implement only a limited set of features (e.g., they address either the downloading of the firmware or its installation) or offer only minimal protection against tampering attacks (e.g., they do not perform a comprehensive verification of the new firmware). Moreover, current solutions are often tailored to a specific hardware and OS [25]–[28] or require the use of specific networking protocols [29], which makes them neither portable nor generic, as discussed next.

Hardware-specific solutions. Several lightweight update systems are provided directly by the hardware manufacturer (e.g., Nordic's BLE Device Firmware Update (DFU) [25] and Texas Instruments' Over-the-Air Download (OAD) ecosystem [26]). Although these solutions are quite popular, they are typically bound to a specific hardware (HW) platform and their implementation is partially closed-source. This precludes portability and practically forces developers to trust a black-box implementation, while preventing further customization.

Solutions with limited capabilities. Operating systems specifically designed for constrained IoT devices (e.g., TinyOS [30] and Contiki [31]) often embed or can be extended with over-the-air reprogramming capabilities. Whilst portable across several HW platforms, many of the existing solutions focus only on a portion of the update process, or do not perform a proper verification of the downloaded firmware and hence cannot ensure its integrity. For example, Contiki-NG [32] allows to download a new firmware using LwM2M[1], but does not embed a bootloader for its installation. Sparrow [27] provides Contiki with over-the-air reprogramming using a custom update agent and bootloader, but – similar to TinyOS' update system [33] – only verifies the CRC code of the downloaded firmware, which is insufficient to protect against tampering attacks.

Combination of independent tools. A few OS for constrained IoT devices, such as RIOT [34] and Zephyr [35], allow to combine open-source solutions taking care of downloading the firmware (e.g., LwM2M and `mcumgr`[2]) with others performing its verification and installation (e.g., `mcuboot`[3]). Whilst the combination of these tools represents, to date, the state-of-the-art when it comes to update solutions for constrained IoT devices, it entails several limitations, as we point out in Sect. II. Combining tools that are designed independently and do not follow a common standard, indeed, results in modules operating in isolation from each other. When coupling together `mcumgr` with `mcuboot`, for example, the verification of the downloaded firmware relies solely on the bootloader. On the one hand, this cannot prevent an attacker from sending an outdated update image that contains well-known vulnerabilities (*update freshness problem*). On the other hand, the verification of a new image takes place only after rebooting the device, which unnecessarily increases the energy expenditure in case of invalid (manipulated) firmwares.

The need for a portable update framework. This state of affairs represents a major problem, as existing solutions for constrained IoT devices are often incomplete, hardware-specific, and – to some extent – insecure. This raises the need of an update *framework* for constrained IoT devices that: (i) takes care of the update process in its entirety, (ii) moves away from the combination of multiple independent tools, and (iii) employs a novel, generic architecture guaranteeing a secure and efficient verification of new firmware. Such a comprehensive update framework should be *portable* across different OS and HW platforms, as well as *open-source*, which would simplify maintenance and customization, as well as prevent vendor lock-in. All these goals should be achieved while maximizing the *energy-efficiency* of the solution (in order to preserve the limited energy budget of battery-powered smart objects), and *without loss of generality*. For example, one should be able to perform updates using both a *push* approach (e.g., by having a smartphone informing the smart object about the availability of new firmware [25], [29]), and a *pull* approach (e.g., by having the smart object periodically polling a server to look for updated software [27], [28]). Likewise, the sought framework should allow to verify the integrity of a new firmware not only once the latter reaches the bootloader, but also after its download (i.e., enabling an early rejection of invalid software).

Our contributions. In this paper we present UpKit, a software update framework for constrained IoT devices that achieves all these goals. UpKit provides a single, lightweight solution encompassing all phases of the update process: from the generation and signature of a new firmware, to the transmission of the latter to an IoT device, its verification and installation. UpKit adopts a novel update architecture that is agnostic to how new firmware images are distributed (i.e., the framework seamlessly supports both push and pull approaches). It further introduces a double-signature process in which a unique request token is included in the manifest of the update image and is signed by the update server. This, together with an additional verification step in the update agent, guarantees the freshness of a new firmware and allows to reject invalid software at an earlier stage, preventing an unnecessary device reboot. UpKit allows sharing of cryptographic libraries between the update agent and the primary device application, which reduces its memory footprint and makes the framework suitable also for highly-constrained devices. Furthermore, UpKit supports differential updates without requiring extra flash space by including a configurable pipeline that allows modifying the received update before storing it in persistent memory. Moreover, in addition to the static update mode (where the firmware is loaded from a fixed position in memory), UpKit supports the A/B update mode alternating two slots for loading and storing the new firmware, which reduces the time required to perform the update and maximizes energy-efficiency.

We have designed UpKit to be portable across different operating systems and hardware platforms, as well as kept its design generic and open-source[4]. To showcase its functionality, we have developed an implementation for three popular OS (Contiki-NG, RIOT, and Zephyr), off-the-shelf hardware platforms (Nordic Semiconductors' nRF52840 plus Texas Instruments' CC2650 and CC2538), as well as cryptographic libraries (TinyDTLS, tinycrypt, and CryptoAuthLib).

An experimental evaluation highlights UpKit's small memory footprint and high energy-efficiency, as well as provides a

[1]Lightweight machine-to-machine (LwM2M) is an efficient protocol developed by the Open Mobile Alliance that is designed, among others, to manage and provide software updates for constrained IoT devices [28].

[2]MCU Manager (`mcumgr`) is an open-source tool allowing communication with remote devices using Bluetooth Low Energy or a serial interface [29].

[3]MCUboot (`mcuboot`) is an open-source bootloader for 32-bit MCUs [36].

[4]https://github.com/updatekit/upkit

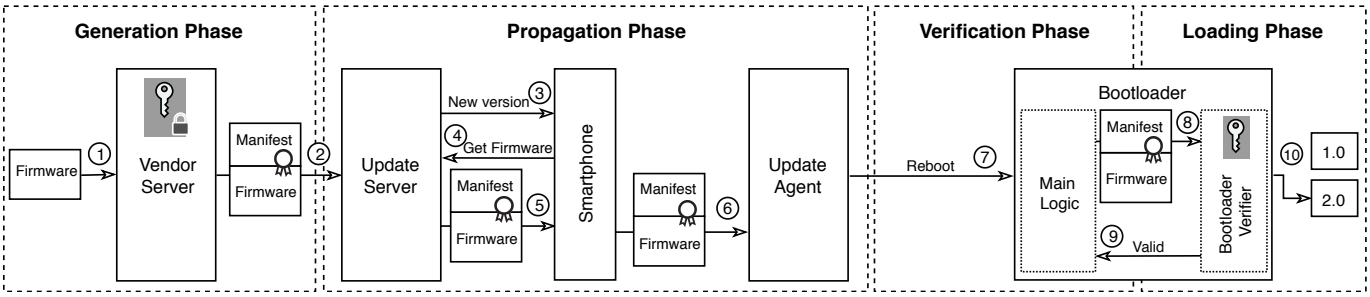


Fig. 1: Commonly-used firmware update architecture for constrained IoT devices.

comparison with popular SW update solutions for constrained IoT devices such as `mcuboot`, `mcumgr`, and LwM2M.

After analyzing in detail the drawbacks of state-of-the-art portable software update approaches for constrained IoT devices in Sect. II, this paper makes the following contributions:

- We present UpKit, a novel framework encompassing all phases of the update process and performing verification of a new firmware also in the update agent. By doing so, UpKit increases the security of the update process, guarantees the freshness of a new firmware, and rejects invalid software at an earlier stage (Sect. III).
- We keep UpKit's design modular, generic, and agnostic to how new firmware images are distributed. Furthermore, we maximize the code reuse of different cryptographic libraries and support differential updates in order to maximize energy efficiency (Sect. IV).
- We design UpKit to be portable and develop a full-fledged implementation for several constrained hardware platforms, OS, as well as cryptographic libraries (Sect. V).
- We evaluate UpKit experimentally in terms of memory footprint and energy-efficiency, providing a comparison with state-of-the-art solutions (Sect. VI).

After describing related work in Sect. VII, we conclude our paper in Sect. VIII, along with a discussion on future work.

II. Limitations of Existing Approaches

Only a few software update solutions for constrained IoT devices are portable across different OS and hardware platforms. Among them, the most widespread are LwM2M, `mcumgr`, and `mcuboot` [28], [29], [36]. As these solutions are designed independently (i.e., without following a common standard) and do not cover the whole update process (LwM2M and `mcumgr` only take care of the distribution of the firmware, whilst `mcuboot` performs its verification and installation), their combination is required in order to assemble a complete update system. In state-of-the-art solutions (e.g., in RIOT and Zephyr), this results in the architecture shown in Fig. 1.

The software update process can be divided into four phases, shown as dashed rectangles in Fig. 1. The firmware is created in the *generation phase* and distributed to the device(s) in the *propagation phase*. Thereafter, the *verification phase* ensures the validity of the received update, and the *loading phase* finally takes care of installing and executing the update.

We describe next these four phases and illustrate the typical update procedure, using a push approach employing `mcumgr` (to propagate the update using a smartphone) and `mcuboot` (to verify and install the firmware) as running example. Note that the same procedure is followed by a pull approach combining LwM2M and `mcuboot`, with the difference that no smartphone is included in the architecture shown in Fig. 1, i.e., there is no gateway between IoT device and update server.

Generation Phase. The update process starts with the vendor server, which embeds a private key to perform digital signature, receiving the firmware binary (1). The vendor server then creates the update image by attaching the manifest (i.e., a set of metadata describing the firmware, such as version and firmware digest) and the generated digital signature. Popular solutions [25], [36] enforce the use of a public-key digital signature, hence excluding symmetric-key algorithms that would make all devices sharing such key vulnerable if the key is compromised [3]. When the update image has been generated, it can be transferred to an update server (2), which is in charge of propagating it to the IoT device(s).

Propagation Phase. The update server, which may be owned by an external company or by the vendor itself, advertises the presence of a new update image version (3). The application running on the smartphone requests the new update image to the server over a secure channel on the Internet (4), and stores the received image locally (5). The smartphone then pushes the update image to the *update agent* running on the IoT device over a secured BLE connection (6). The update agent is a portion of the firmware running on the IoT device that is used to interact with the update server for downloading a new update image, as well as for storing the latter in persistent memory. During the propagation phase, the *freshness of the update image* must be granted, which consists in preventing that outdated update images reach the device, i.e., precluding an attacker from sending a valid, but old image with well-known vulnerabilities. Popular solutions such as `mcumgr`, do not provide any solution to mitigate this issue, whilst LwM2M, instead, relies on transport layer security to grant update freshness. In fact, when considering a configuration using LwM2M and `mcuboot`, the freshness of an update is granted by the secure connection between the server and the update agent, which is carried out using secure CoAP with public-key based client authentication. This, however, implies a *direct* connection between server and IoT device, which, in

many cases (e.g., when using a gateway or a smartphone, as in Fig. 1), is not available. Once the update image has been fully stored on persistent memory, the IoT device reboots (7) and the update process is resumed by the bootloader. Note that state-of-the-art solutions such as `mcumgr` and LwM2M do not make use of differential updates. The latter could enhance the efficiency of the propagation phase by reducing the amount of data to be transferred and stored on the IoT device.

Verification Phase. Before loading the new update image, the bootloader checks its digital signature to verify its integrity and authenticity. In case the verification fails, the bootloader must remove the new update image and perform the rollback to the previous one. If the verification is successful, instead, the installation proceeds (9). Note that postponing the verification to the bootloader has a negative impact on the energy-efficiency of the solution. Indeed, if an invalid update is received (e.g., tampered while stored on the smartphone or modified during transmission if an untrusted channel is used), the IoT device still needs to download and store the full update in memory, as well as to reboot – even though one may already recognize the new update as invalid from the manifest. This unnecessary download and reboot of the IoT device reduces not only its functionality, but also its availability, as rebooting the device causes its temporary disconnection from the network.

Loading Phase. The update image typically contains the full application statically linked to a specific memory offset. This allows the update agent to locate the image on any memory object, while the bootloader moves the update image to the right offset in order to correctly load the firmware. If the verification was successful, the bootloader moves the update image to the correct memory address and jumps to it, so to start executing the new firmware (10).

Key limitations. The architecture depicted in Fig. 1, which is commonly adopted when updating the software of constrained IoT devices, presents a number of fundamental problems:

- The freshness of an update (i.e., the rejection of outdated images) relies exclusively on the security of the connection with the server, which in case of LwM2M consists in transport-layer security. Unfortunately, an end-to-end secure channel between IoT device and update server is not necessarily available, especially in the presence of intermediary devices such as gateways and smartphones, as discussed previously. Therefore, the update freshness property must be granted during the verification phase as already done with the integrity and authenticity properties, independently from the network protocol used.
- The update agent is only designed to propagate the new firmware and does not verify the validity of the update image. Hence, an invalid firmware may be downloaded and stored on the IoT device, unnecessarily increasing its energy expenditure. Furthermore, during reboot, the IoT device may not be reachable for an extended amount of time, which affects the availability of the system.
- Most IoT systems do not embed any functionality to update the bootloader [3], [35]. The latter is indeed a complex task that could brick the device in case of failure (i.e., leave the device in an inconsistent state, making it useless). This implies that bugs or vulnerabilities in the verification of the firmware (e.g., the use of deprecated digital signatures and digest algorithms) cannot be mitigated. This raises the need to perform verification also on the update agent: while the bootloader would still remain vulnerable, no invalid update image is forwarded to it.

All these issues can be solved by improving and anticipating the verification of new firmware to the update agent, as well as by providing a *single* solution covering the whole update process. To this end, we have designed UpKit: a SW update framework for constrained IoT devices that we describe next.

III. UpKit: General Architecture

UpKit is a portable update framework for constrained IoT devices encompassing all phases of the update process and supporting several features. In contrast to the state-of-the-art approaches described in Sect. II, UpKit introduces a verification step also in the update agent and makes use of a novel architecture, which is shown in Fig. 2.

A. Generation Phase

Also in UpKit the update process starts with the vendor server, receiving the raw firmware binary ❶, which is then used to generate the update image by adding the manifest and its digital signature. The update image is then loaded on an update server that is connected to the Internet and is reachable from any IoT device ❷. However, differently from common architectures (Fig. 1), in UpKit the update server is also involved in the generation phase. In particular, the update server performs a *double signature* on the update image to ensure that an update is bound to a specific device and request. This allows to grant update freshness during the verification phase, without the need to rely on transport layer security.

B. Propagation Phase

Once a new update image is generated, the update server announces its availability over the Internet ❸. Smartphones acting as proxy receive this information and use their local connection to the IoT device(s) to request a *device token* ❹. The latter is a structure containing the following fields:

- a 32-bit unique ID of the device, which may be derived from unique identifiers such as the MAC address;
- a 32-bit nonce generated by the device for each request;
- a 16-bit value containing the current version if differential updates are supported by the device, or zero otherwise.

The device token is received by the smartphone ❺, which sends it to the update server while requesting the new available update image. The values of the device token are then added to the manifest and signed again: this makes the update *unique* for each device and each request. This way, the IoT device knows that the update image is the last one available on the server, provided that: (i) the device token values match the previous ones and (ii) the update server's signature is valid.

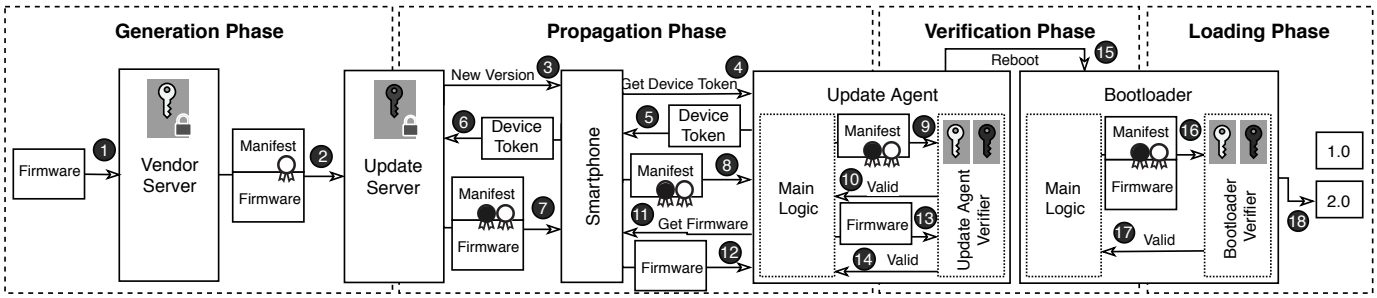


Fig. 2: UpKit's architecture: in contrast to existing solutions, UpKit introduces a double-signature process and performs verification also in the update agent, which guarantees the freshness of a new firmware and allows an early rejection of invalid SW.

To ensure update freshness, we have also considered other approaches, such as the inclusion of a timestamp in the manifest indicating the expiration time of the update image [10]. However, we excluded this approach, as it requires a reliable time source on each IoT device. Indeed, time sources may easily be faked, as shown by attacks against the NTP protocol [37], [38]. Furthermore, the use of timestamps does not permit to block the installation of an update until the timestamp expires, which may allow attackers to install an update with known vulnerabilities even though a new update version is available. The double signature approach proposed by UpKit, instead, ensures that the *latest* update image is downloaded, and only assumes that the private key stored in the update server remains secret, i.e., UpKit does not require a reliable time source or a real-time clock on each IoT device.

The update server uses the *current version* value in the device token to decide whether differential updates should be used[5]. If differential updates are supported, the update server uses the current version to derive the delta with the latest update image. The full or differential update is then signed by the update server and sent to the smartphone ❼.

Once it fully received the update image, the smartphone starts forwarding the manifest to the update agent running on the IoT device ❽. As soon as the manifest has been received, the update agent verifies its content ❾ and stops the update process if any of the values or the digital signatures are invalid. Otherwise, if the validation process is successful ❿, the update agent notifies the smartphone that the firmware can be transmitted ⓫. Once the firmware reached the IoT device ⓬, it undergoes a first verification phase ⓭ ensuring that the calculated digest is equal to the one included in the manifest previously received. If this is the case ⓮, the device can reboot to load the new firmware ⓯.

Compared to the state-of-the-art software update architectures shown in Fig. 1, UpKit introduces a first verification step in the update agent. This ensures that the update image is valid, granting (i) integrity and authenticity by means of the vendor server's signature, and (ii) update freshness by means of the update server's signature. This allows the IoT device to reject invalid software at an early stage, reducing the communication overhead to a bare minimum and avoiding an unnecessary reboot. Furthermore, as the update agent is a portion of the firmware running on the device (and is hence embedded in new update images), it can be updated in case some bugs or vulnerabilities arise within the verification procedure.

The architecture shown in Fig. 2 illustrates a push approach and includes a smartphone to forward the updates to the IoT device(s). Note, however, that the smartphone is not an active component of UpKit, as it does not modify the update image in transit. UpKit also supports updates that follow a pull approach by allowing a direct connection between the update agent and update server. Indeed, every device in between these two, being it a smartphone or a gateway (border router), is only in charge of forwarding the update image, and has no active role in the update process. This implies that even a compromised gateway does not affect the integrity and authenticity of an update, nor the ability of UpKit to guarantee the freshness of a new image, which is granted by the signature performed on the update server. Still, a compromised gateway can perform attacks against the device, such as denial of service attacks or prevent an update from reaching the update agent. However, these attacks are not strictly related to UpKit, but affect any update system involving a device acting as proxy.

C. Verification Phase

In contrast to common approaches performing verification only in the bootloader (Fig. 1), UpKit already performs a first verification in the update agent to ensure that every new firmware stored on the device memory is valid. Nevertheless, the verification on the bootloader is still important, since the one executed by the update agent may not be sufficient to ensure the integrity and authenticity of the update image. For example, the IoT device may reboot in the middle of the propagation phase, which would leave the new update image stored on the device incomplete. Similarly, the device may lose power before the update agent can verify the firmware. To prevent this, the bootloader must ensure the validity of the values contained in the firmware's manifest and digest, as well as of its digital signatures after reboot ⓰. Only if the update is valid ⓱, the bootloader proceeds to the loading phase, during which the new update is effectively loaded on the device.

[5]A device can indicate to the server that is unable to support differential updates by including a zero value in the current version field. The use of differential updates, indeed, increases the memory usage of the update agent.

D. Loading Phase

The loading phase is the last step before the new code is executed on the IoT device. It consists in preparing the image to be executed, such as moving it to the right memory address ⑱. Similarly to other update solutions for constrained IoT devices [3], [35], also UpKit does not support updating the bootloader, as any failure during this phase would be fatal to the system and brick the device. However, as discussed earlier, any bug in the bootloader's verifier can be mitigated by updating the verifier contained in the update agent, thus preventing an invalid image to reach the bootloader itself.

IV. UpKit: Inner Workings

The core of UpKit lies in the *update agent* and the *bootloader* running on the IoT device, which orchestrate the propagation, verification, and loading of new firmware. As described next, we design both components to be highly modular (Sect. IV-A) and agnostic to the network configuration used to distribute a new firmware image (Sect. IV-B), which makes UpKit easily portable and maximizes code reuse. We also design UpKit's modules to retain a high efficiency, e.g., by means of a configurable pipeline to manage differential updates, as well as a flexible memory implementation to customize memory slots and support A/B updates (Sect. IV-C). Finally, we enable a double verification by including a *verifier module* in both bootloader and update agent (Sect. IV-D).

A. Modular Design

UpKit's modules are organized in a three-layer structure, as shown in Fig. 3, where each layer depends on the underlying one for its operation. *Common modules* are designed to work independently from the OS or the hardware platform, and rely on *common interfaces* to interact with low-level functions, such as flash memory access as well as network or cryptographic implementations. An implementation of each interface is provided by *platform-specific modules*, which allows to handle several hardware platforms and OS. The latter, indeed, can exhibit a large heterogeneity of memory implementations and cryptographic libraries, as discussed in Sect. V.

A *finite-state machine (FSM) module* coordinates the operations of the update agent, managing the data received by a pull or push connection. This data is passed to a *pipeline module*, which modifies it before storing it to persistent memory. The *memory module* provides functions to handle the stored data, whilst the *verifier module* takes care of the double verification in both update agent and bootloader, as described in Sect. III.

B. Support for Different Network Configurations

UpKit is agnostic to the network configuration used to distribute a new firmware, which ensures generality. To this end, a FSM coordinates the update process, independently from the use of a pull or push connection. When using the former (e.g., a CoAP connection to the server), the update agent performs requests to the server and passes the received data to the FSM. Similarly, when using the push approach (e.g., a smartphone connecting via BLE to the device), the data received from the callback function is passed to the FSM, which handles the received data according to the reached state.

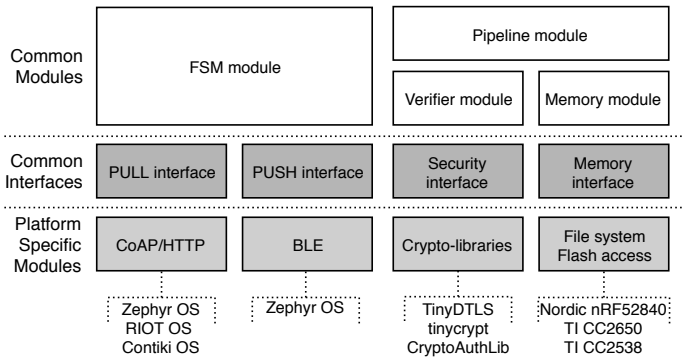


Fig. 3: UpKit's design is highly portable: *common modules* and *interfaces* allow to keep *platform-specific* code separate, so to handle heterogeneous hardware platforms and OS.

The FSM moves across eight states, as shown in Fig. 4:

- *Waiting*. The update process is idle until a device token is requested to the FSM. The device token is filled with a nonce, the device ID, and the current version. The nonce is stored in the FSM state to be compared with the received one, granting *update freshness*. The FSM returns the device token and moves to the next state.
- *Start update*. The memory slot containing the oldest firmware (i.e., the one with the smallest version number) is erased to make arrangements for the new update image.
- *Receive manifest*. The FSM accept data until the size of the manifest is reached and then moves to the next state.
- *Verify manifest*. The verifier module of the update agent ensures the manifest validity by checking its digital signatures to grant integrity and authenticity. Furthermore, it ensures that the other manifest values are correct, e.g., by comparing the received device ID and the nonce with the one of the device token, as well as by ensuring that the new version is higher than the last one available. If the manifest is valid, the FSM moves to the next state.
- *Receive firmware*. The FSM accepts chunks of the firmware until its size, contained in the manifest, is received. The FSM then moves to the next state.
- *Verify firmware*. The digest of the received firmware is calculated and compared with the digest contained in the manifest. If the two digests match, the integrity of the update is granted and the propagation phase is completed. The device can then reboot to apply the update.
- *Cleaning*. This state is reached if the verification is unsuccessful or in case of other errors. All variables in the FSM are initialized and the used slot invalidated.

The FSM does not directly write data to persistent memory. Instead, it passes the data to a pipeline module, which transforms it on-the-fly before storing it in memory. This allows to easily support differential updates without requiring additional memory slots, as described in Sect. IV-C.

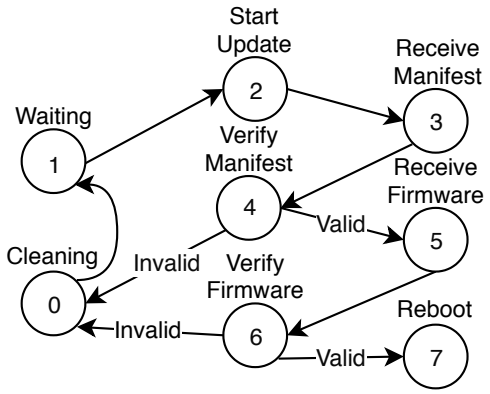


Fig. 4: UpKit's finite-state machine (FSM).

C. Support for Efficient Operations

UpKit allows an efficient software update process by supporting *differential updates* (which reduce the size of the update to be transferred) as well as by supporting *A/B updates* (which reduce the time required for the loading phase).

Differential updates. After receiving the device token (see Fig. 2), the update server generates a patch containing only the differences between the firmware's current version and the new update image. The resulting patch is smaller in size compared to a full firmware image and can be transferred in a more efficient way to the IoT device, hence minimizing the amount of time during which its radio transceiver is turned on. Instead of storing the patch in memory before applying it, in UpKit we implement a pipeline able to apply the patch on-the-fly, i.e., writing directly the new update on persistent memory. This avoids the need of an additional memory slot to store the patch, which would reduce the space available for the actual application code. Before being applied, the patch goes through a *decompression* stage and a *patching* stage.

We implement these two stages based on the results of Stolikj et al. [19], who identify the `bsdiff` and `lzss` algorithms as the ones offering the best compromise between size of the generated patch and memory footprint of the patching and decompression routine.

UpKit's pipeline includes four stages, as shown in Fig. 5:

- *Decompression stage*: the delta generated on the server needs to be decompressed on the device using `lzss`, an improved version of the popular `lz77` algorithm with a small usage of flash memory and RAM [19].
- *Patching stage*: once the data is decompressed, the output is a patch that needs to be applied to the latest firmware in order to generate the newest. The patching stage is implemented using the `bspatch` routine, which performs the opposite of the `bsdiff` algorithm used on the server.
- *Buffer stage*: input data is stored in a buffer until the latter is full. Matching the buffer size with the flash sector size results in faster writes and fewer flash erasures.
- *Writer stage*: this is the last stage of the pipeline, which interacts directly with the memory interface to write the data received from the buffer to persistent memory.

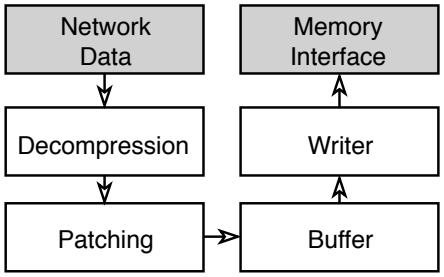


Fig. 5: Pipeline stages.

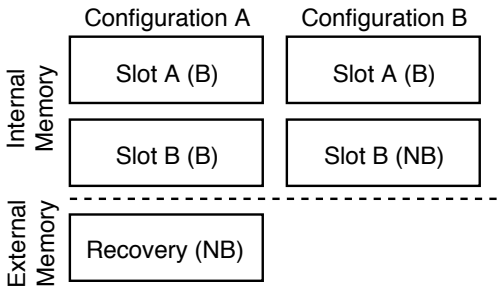


Fig. 6: Memory slots configurations.

Flexible memory slots. In order to support SW updates, the persistent memory of an IoT device needs to be organized in slots, each one storing a single update image. The organization of these slots depends on the type of memory available (i.e., one or more flash memories), and on the loading mode of the bootloader (i.e., on the number of bootable slots available). In UpKit, slots are managed by the *memory module*, which provides portable functions to erase, swap, or override a slot with another. To support multiple flash memories, several slots, and different loading modes, UpKit defines two types of memory slots: bootable slots (B), which contain a directly executable image, and non-bootable slots (NB), which require the image to be moved to a bootable slot to be executed. This distinction allows to easily support different configurations, as shown in Fig. 6. UpKit supports both *static* software updates, where there is only one bootable slot (Fig 6., Configuration B), as well as *A/B updates*, where two bootable slots are available, allowing the bootloader to directly jump to the newest slot without having to swap the images (Fig 6., Configuration A).

D. Support for Double Verification

To implement verification both in the bootloader and the update agent, UpKit uses the same *verifier module*. The latter handles the security features described in Sect. III, ensuring that the digital signatures of the firmware digest and the manifest are valid. Moreover, the *verifier* module grants the compatibility of the update image with the device characteristics, by ensuring the validity of the manifest fields:

- *ID*: indicates a unique device identifier. It is needed to grant update freshness and the received update image must contain the same ID included in the device token.
- *Nonce*: indicates a unique number that refers to the firmware request. It is needed to grant update freshness,

and it must match the one included in the device token.
- *Old version*: indicates the version to which the differential update has been calculated by the update server;
- *Version*: each received update image must have a version strictly higher than the newest one available on the device.
- *Size*: indicates the size in bytes of the firmware and allows to block the reception in case it has been exceeded.
- *Digest*: represents the hash of the firmware and ensures its integrity once it has been received.
- *Link offset*: indicates the memory address for which the firmware has been built to ensure that a firmware is compatible with a specific slot;
- *App ID*: a unique identifier indicating the application and HW platform for which the update has been built.

Compared to the manifest used by existing solutions (e.g., `mcuboot` and `mcumgr`), UpKit's manifest adds the first three fields (i.e., *ID*, *nonce*, and *old version*) as well as the update server's signature. These fields allow to grant update freshness independently from the network configuration (see Sect. III) and allow to support differential updates (see Sect. IV-C).

V. Implementation

We implement UpKit for a wide range of constrained IoT hardware platforms (such as Nordic Semiconductors' nRF52840, Texas Instruments' CC2650 and CC2538), as well as operating systems (such as Contiki, RIOT, and Zephyr). Furthermore, we include the support of two cryptographic libraries (`TinyDTLS`, `tinycrypt`), as well as support for a hardware security module (HSM) family. We detail next the implementation of update agent and bootloader for the aforementioned platforms, OS, and cryptographic implementations.

UpKit's update agent. The update agent requires *all* the *common modules* depicted in Fig. 3. We therefore implement the four *common interfaces* according to the employed HW platform (and its memory interface), cryptographic library (SW implementation or HW crypto-modules), and network configuration (push or pull interface). We base the pull interface on the CoAP implementation provided by each OS (i.e., `Zoap` for Zephyr, `libcoap` for RIOT, and `er-coap` for Contiki) and test our pull implementation on the TI CC2650 with Contiki, the TI CC2538 with RIOT, and the Nordic nRF52840 with Zephyr. For each platform, we have configured an IEEE 802.15.4 network with a border router exposing a test server deployed on an IPv6 network. We implement the push interface on the nRF52840 board with Zephyr, as it offers complete support for BLE GATT. While also Texas Instruments provides a BLE stack for the TI CC2650, the Contiki OS used to test this platform does not fully support it and we hence leave its implementation as future work. To test our push implementation, we have developed an iOS application using the Swift programming language that allows sending the update to the device using BLE. We have also packaged the main functions used to interact with the device in an SDK simplifying the development of new iOS applications.

UpKit's bootloader. We build UpKit's bootloader on top of Zephyr, RIOT, and Contiki. Compared to a bare-metal implementation, the reliance on an OS increases the size of the bootloader, but, at the same time, grants a high level of portability. Among others, the existence of an underlying OS allows an easy port of the bootloader to all the HW platforms supported by the OS, without the need to manage platform initialization code or to support various memory drivers. Out of the modules presented in Fig. 3, UpKit's bootloader only needs access to the memory and the verifier module. The former relies on the *memory interface*, which abstracts the flash memory details to the upper levels. The API is inspired by the standard POSIX IO functions, allowing to open and close a memory slot, as well as to read and write data. To support flash memories and the need of sector erase before writing, specific *open modes* have been defined: `READ_ONLY` allows to only read the memory, `WRITE_ALL` erases all the content of a slot to allow writing continuously, and `SEQUENTIAL_REWRITE` automatically erases each new page encountered. The memory interface also allows assigning a Linux file to each slot, which gives the ability to work with devices supporting a file system, as well as to test the library modules without the need of a simulator. We developed an implementation for each HW platform, using a device driver for the TI CC2650 and CC2538, as well as Zephyr's flash abstractions for the nRF52840. As the internal memory of the TI CC2650 is insufficient to store both memory slots, we use its external flash to store the non-bootable slot. Support for internal and external memory is granted by a structure of functions pointers that allows to use custom memory functions depending on the type of flash used.

The *security interface* abstracts the security features required by UpKit, allowing to use different cryptographic libraries, as well as to exploit cryptographic HW accelerators included on the chosen platform. After testing a number of cryptographic libraries (including, among others, `polarssl`, `matrixssl`, `wolfssl`, and `libtomcrypt`), we have implemented UpKit with `TinyDTLS`, `tinycrypt`, and `CryptoAuthLib`. These libraries were the ones exhibiting the smallest memory footprint, while supporting ECDSA signature verification using the *SHA-2* and the *secp256r1* ECC curve. Note that the `TinyDTLS` and `tinycrypt` implementations perform the verification in software, while the `CryptoAuthLib` allows to work with the ATECC508 HSM [39] (Atmel's CryptoAuthentication). We use the latter in conjunction with the TI CC2650 platform to safely store public keys, preventing external actors from modifying them, as well as to perform signature verification in hardware, which also reduces the amount of flash required.

VI. Evaluation

We evaluate UpKit experimentally by analyzing the memory footprint of its components running on the IoT device, namely the *bootloader* and the *update agent*. In particular, we measure UpKit's memory footprint across different OS and HW platforms, as well as different crypto-libraries (Sect. VI-A).

We then compare UpKit's memory footprint with state-of-the-art software update solutions for constrained IoT devices, such as LwM2M and `mcumgr` as update agent for the pull and push approach, as well as `MCUboot` as bootloader (Sect. VI-B). We finally analyze the time necessary to complete an update when using different UpKit's configurations (Sect. VI-C). Note that the evaluations carried out in this section are obtained by disabling all debugging and logging features of UpKit and other solutions, in order to ensure a fair comparison. The same applies to all logging and debugging features of the operating systems, which have been disabled to reduce as much as possible the size of the generated firmware.

A. UpKit's Memory Footprint

We begin our evaluation by comparing the amount of flash memory and RAM used by UpKit's bootloader and update agent for the different implementations discussed in Sect. V.

UpKit's bootloader. Table I summarizes the memory footprint of UpKit's bootloader built with Zephyr, RIOT, and Contiki, when using different cryptographic libraries. The flash usage is comparable across different OS when using the same crypto-library, with the Zephyr build requiring about 15% less flash memory than the one of other OS. Zephyr, however, requires roughly 20% more RAM due to its larger run-time stack. Table I also shows that the UpKit bootloader built with `TinyDTLS` requires around 1.10 kB less flash memory than the one built using `tinycrypt`, regardless of the employed OS. We have also evaluated UpKit's memory footprint when using Contiki and the `CryptoAuthLib` library to connect to a ATECC508 HSM, as discussed in Sect. V. With this configuration, the bootloader requires only 14078 bytes of flash memory, i.e., about 10% less flash memory than the bootloader built based on Contiki and using `TinyDTLS`. UpKit's bootloader's code is highly portable: for each platform, approx. 91% of the code is platform-independent. The remaining 9% of the code is platform-specific and deals with the management of the flash memory drivers, as well as the interrupt vector table to store and execute the update image.

UpKit's update agent. Table II compares the memory footprint of UpKit's update agent when using different OS and network configurations (i.e., pull or push approach), as well as `TinyDTLS` as a cryptographic library. When using a pull approach based on CoAP, Contiki exhibits the smallest build for the update agent, both in terms of flash and RAM usage. In particular, Contiki uses 64% and 17% less flash memory as well as 73% and 36% less RAM than Zephyr and RIOT, respectively[6]. We further evaluate the memory footprint of UpKit when using a push approach based on our Zephyr implementation described in Sect. V. This build image makes use of roughly 82 kB of flash and 21 kB of RAM, much less than the Zephyr build for the pull approach. Indeed, when using the pull approach, the full IPv6 stack and the CoAP library need to be included, whereas only the BLE stack is required when using the push approach, which results is a lower memory footprint for the latter. In addition to the network stack, the *pipeline* and *memory modules* require a significant portion of flash (1632 and 2024 bytes, respectively). This is mostly due to the differential patcher (`bspatch`) and the decompression (`lzss`) algorithms for the *pipeline module*, as well as to the functions to copy and swap two memory slots for the *memory module*. The *pipeline* module also requires a non-negligible amount of RAM (2137 bytes), due to the space allocated to the `lzss` buffer. UpKit's update agent is highly portable: in average, only 23.5% of the code is platform-specific. In contrast to the bootloader, the code of the update agent must not only manage the flash memory to store the update image, but needs also to interact with the network stack to download the update image, which results in additional lines of code being platform-dependent.

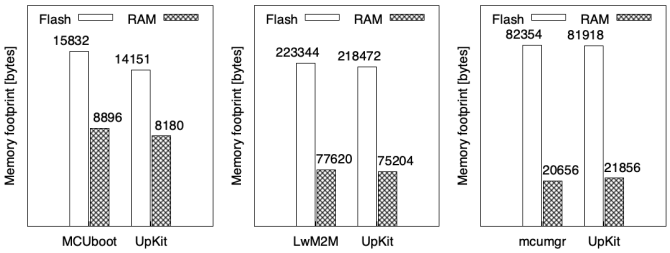
[6]Table II shows the smallest version of UpKit's update agent that could be configured for Zephyr, RIOT, and Contiki. Note that, as discussed in Sect. V, the different OS use different implementations of the CoAP library (i.e., Zoap, er-coap, and libcoap), as well as different underlying layers (i.e., 6LoWPAN or OpenThread), which results in largely different memory footprints.

Operating System	Library	Flash	RAM
Zephyr bootloader	TinyDTLS	13040	8180
	tinycrypt	14151	8180
RIOT bootloader	TinyDTLS	15420	6512
	tinycrypt	16552	6512
Contiki bootloader	TinyDTLS	15454	6637
	tinycrypt	16546	6637
	CryptoAuthLib	14078	6553

TABLE I: Memory footprint of UpKit's bootloader.

Approach	Operating System	Flash	RAM
Pull (6LoWPAN)	Zephyr	218472	75204
	RIOT	95780	31244
	Contiki	79445	19934
Push (BLE)	Zephyr	81918	21856

TABLE II: Memory footprint of UpKit's update agent.



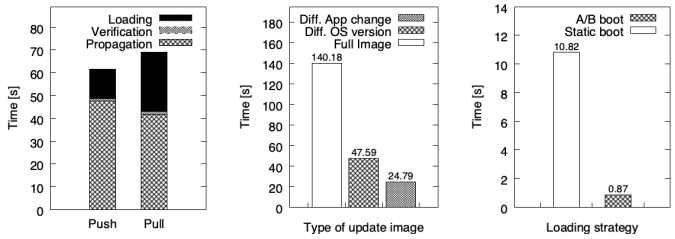
(a) Bootloader (b) Pull approach (c) Push approach

Fig. 7: The RAM and flash memory footprint of UpKit is comparable to or lower than state-of-the-art solutions.

B. Comparison to Existing Solutions

We compare next the memory footprint of UpKit with the one of state-of-the-art solutions such as LwM2M, `mcumgr`, and `mcuboot`, showing that UpKit always requires less flash memory, while improving the security and efficiency of the update process. We perform the comparison using the Zephyr OS and the nRF52840 platform, since it is the only one that supports all the aforementioned state-of-the-art solutions.

Fig. 7a shows the comparison between UpKit's bootloader and `mcuboot`. Both have been configured to use the ECDSA



(a) Push vs. pull (b) Diff. updates (c) A/B updates

Fig. 8: Time required to complete a SW update using UpKit.

algorithm (*secp256r1* ECC curve and *SHA-256* digest algorithm) with the `tinycrypt` library. Our results show that the UpKit bootloader requires 1600 and 716 bytes less flash memory and RAM than `mcuboot`, respectively.

Fig. 7b compares the memory footprint of UpKit's update agent configured for the pull approach with that of LwM2M. UpKit's update agent requires 4.8 and 2.4 less kB of flash and RAM, respectively. Note that the larger memory footprint of LwM2M is due to its embedded M2M features, which would allow to transmit also other data between device and server. It is important, however, to note that UpKit only focuses on the update process, whilst LwM2M could, in principle, also be used to expose multiple objects to the server, enabling M2M communication. Since the focus of this paper lies on the ability to perform a software update, we disabled the other services of LwM2M to ensure a fair comparison.

Fig. 7c compares the memory footprint of UpKit's update agent configured for the push approach with that of `mcumgr`. To perform the comparison, we disabled all features of `mcumgr` not related to SW updates, such as file system, logging, and OS management features. Our results show that UpKit requires 426 bytes less flash and only 1200 bytes more RAM than `mcumgr`, despite the addition of numerous features, such as differential updates and signature validation.

C. UpKit in Action

We compare next the time required by UpKit to complete the different phases of a SW update using both the pull and the push approach. We further quantify UpKit's increased efficiency thanks to its support for differential and A/B updates.

Push vs. pull approach. We measure the time necessary to propagate, verify, and load a full-image firmware of 100 kB using UpKit running on the nRF52840 with Zephyr. Fig. 8a shows our results and compares the time required to complete an update when using a push and a pull approach to perform a full-image update. The push approach takes 61.5 seconds, 7.6 seconds less than the pull approach: this is due to the size of the image to be swapped (larger for the pull case, see Table II). Therefore, in the push approach, the number of sectors to be swapped between the two memory slots is smaller. Fig. 8a also breaks down the time spent in each update phase. The propagation phase takes most of the time: 47.7 and 41.7 seconds of the overall time for the push and pull approach, respectively. The verification phase is executed in the same way for both push and pull approaches, and accounts for 1.78% and 1.72% of the total update time, respectively. Finally, the loading phase accounts for 20.6% and 37.9% of the total update time for the push and pull approach, respectively. UpKit allows to reduce the time spent in the most time-consuming update phases (propagation and loading) by using differential and A/B updates: we quantify next the benefits introduced by these features.

Efficiency of differential updates. Fig. 8b shows the impact of differential updates on the duration of the update process using the pull approach described above. The figure compares *full-image updates* against two different cases of differential updates: an *OS version change* (e.g., from Zephyr v1.2 to v1.3), and a *change in application functionality* (e.g., 1000 bytes of difference). Fig. 8b shows that the use of differential updates allows UpKit to reduce the overall update time by up to 66% and 82% in case of an OS version and application functionality change, respectively. Note that this time is saved exclusively in the propagation phase, as loading and verification are not performed on the patch, but on the entire update image.

Efficiency of A/B updates. Fig. 8c shows that the use of A/B updates decreases the time required to complete the loading phase by 92% compared to a static boot. This reduction derives from the fact that the bootloader can simply load the new firmware, instead of copying or swapping the slot as required with static updates. Note that this reduction in time is independent on the approach used (pull or push), since the use of A/B updates only affects the loading phase.

VII. Related Work

A large body of work has focused on updating the SW running on a device: from that of standard computers [40], to the over-the-air update of smartphones [41], vehicles [42]–[46], and sensor networks [33]. When it comes to updating IoT devices, existing solutions are vastly different, as they are tailored to the requirements and capabilities of each device.

Updating powerful IoT devices. IoT devices running Linux distributions (e.g., gateway nodes [47], [48]) perform SW updates using three main approaches. *Package-based* approaches (e.g., `YUM` [13] and `APT` [14]) operate on a file-level, which minimizes the update size and grants a high flexibility, but may lead to inconsistencies when only a portion of the necessary modules is updated successfully [8]. Approaches relying on *Linux containers* (e.g., `Balena.io` [18]) allow updating an application and all its dependencies atomically, which represents a good trade-off between the size of an update and the update's consistency. Finally, approaches based on a *full-image update* (e.g., `swupdate` [15], `Mender.io` [16], and `RAUC` [17]) maximize consistency by atomically updating the entire firmware, at the cost of a larger size and lower flexibility. All the aforementioned solutions require a significant amount of memory (tens of Megabytes for both RAM and ROM), which makes it impossible to reuse or even adapt such approaches for constrained IoT devices. Furthermore, the latter typically do not support containers and their OS are typically

not based on a file system, which also makes package-based and container-based approaches not applicable.

Updating constrained IoT devices. For constrained IoT devices, a few vertical solutions exist, such as Nordic's DFU [25] and Texas Instruments' OAD ecosystem [26]. These solutions only work with Nordic and Texas Instruments hardware or on platforms supported by FreeRTOS, and are hence not portable. Similarly, other custom solutions have been specifically tailored to a OS or HW platform, such as Mark Solters's OTA project [49] (focusing on Contiki and the TI CC2650 only), as well as Sparrow [27] (focusing only on Contiki and the Zolertia Zoul Re-Mote). Considering the plethora of OSes targeting constrained devices [20], it is neither feasible nor advisable that each OS maintains its own solution. This indeed results in OS without an update system (e.g., *NuttX* [50]) or with an incomplete one, e.g., Sparrow [27] (used by Contiki), and Deluge [33] (used by TinyOS) only verify the CRC to ensure integrity, which does not protect against tampering. UpKit solves this problem by providing an open-source, portable, and lightweight solution able to securely perform SW updates across different OS and HW platforms.

Portable solutions for constrained devices. LwM2M [28], `mcumgr` [29], and `mcuboot` [36] are portable solutions that can be combined to build an update system for constrained IoT devices. LwM2M is a M2M standard [51] that enables a device to export a firmware object to perform SW updates. Although it is well supported on many OS, (e.g., Zephyr and Contiki), LwM2M only performs the *propagation* of an update, relying on transport layer security to ensure update freshness, and on the bootloader to verify integrity and authenticity. The same limitations apply to `mcumgr` [29], a device management library supported on mynewt and Zephyr that allows downloading an update over BLE or a serial shell. In contrast, UpKit performs a double verification, which grants update freshness and allows to reject invalid SW earlier, thus also preventing unnecessary radio communication and unwanted reboots. Finally, `mcuboot` is a portable bootloader for constrained devices that is compatible with different OS (e.g., Zephyr and RIOT) and that works in combination with `mcumgr` or LwM2M. However, `mcuboot` does not provide any mitigation against vulnerabilities of the bootloader. In UpKit, instead, a double verification process rejects invalid updates already on the update agent. As the latter can be seamlessly updated, one can also protect a vulnerable bootloader by rejecting firmwares that are explicitly crafted to exploit well-known weaknesses of the bootloader's verification procedure. UpKit also makes use of a double signature to (i) prevent that compromising a single signature can lead to the generation of a valid update, and (ii) to ensure update freshness, leaving to the update server's signature the role of signing the update for each device request. A similar approach has been proposed by TUF [40], an update framework for more powerful devices often used as reference for package-based SW update systems. Other approaches grant update freshness by making use of a timestamp in the manifest to indicate the expiration time of the update image [10]. This, however, requires a reliable time source on each IoT device and makes an update valid until a fixed expiration time, even if new update images are available meanwhile. Another category of solutions to update constrained IoT devices in a portable way is provided by Cloud providers. For example, Microsoft Azure [52] offers a library suitable to connect IoT devices to the provider's SW update service. However, whilst these solutions are suitable for the same device classes targeted by UpKit, they require each IoT device to embed a real-time clock or to have the ability to connect to an NTP server for establishing the TLS connection and generating the secure authentication token [53]. In contrast, UpKit does neither require additional HW (which would increase the device cost) nor requires the use of NTP servers (which would expose the system to security vulnerabilities, as discussed in Sect. III).

VIII. Conclusions and Future Work

In this paper, we have presented UpKit, an open-source, portable, and lightweight software update framework for constrained IoT devices. UpKit addresses the architectural limitations of existing solutions and enables the verification of a new firmware also on the update agent. This allows to increase the security of the update process, to guarantee the freshness of a new firmware, and to reject invalid software at an early stage, hence preventing an unnecessary reboot of the device.

We kept UpKit's design modular and agnostic to how a new firmware image is distributed, as well as enriched the framework with features such as support for differential updates and flexible memory slots. This makes UpKit easily portable, efficient, and suitable for constrained IoT devices. After developing an implementation for three popular OS and off-the-shelf hardware platforms, we have evaluated UpKit's memory footprint and compared it with that of state-of-the-art solutions.

Future work includes the port of UpKit to additional OS and the support of the upcoming IETF SUIT standard [10], in order to allow inter-operation with a larger range of IoT solutions. We further plan to add a decryption stage in UpKit's pipeline module, in order to make confidentiality independent from the employed transport security layer.

Acknowledgments

This work has been performed within the LEAD project "Dependable Internet of Things in Adverse Environments" funded by Graz University of Technology. This work was also partially funded by the SCOTT project. SCOTT (http://www.scott-project.eu) has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This joint undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, Norway. SCOTT is also funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between May 2017 and April 2020. More information at https://iktderzukunft.at/en/.

References

[1] A. Al-Fuqaha et al. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.

[2] M. Ersue, D. Romascanu, J. Schoenwaelder, and U. Herberg. RFC 7547: Management of Networks with Constrained Devices: Problem Statement and Requirements. IETF, 2015.

[3] E. Ronen, A. Shamir, A. Weingarten, and C. O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *Proc. of the IEEE Symposium on Security and Privacy (SP)*, pages 195–212. IEEE, 2017.

[4] M. Farooq, M. Waseem, A. Khairi, and S. Mazhar. A Critical Analysis on the Security Concerns of Internet of Things (IoT). *International Journal of Computer Applications*, 111(7), 2015.

[5] K. Angrishi. Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets. *arXiv preprint 1702.03681*, 2017.

[6] F. Acosta-Padilla, E. Baccelli, T. Eichinger, and K. Schleiser. The Future of IoT Software Must be Updated. In *Proc. of the Intl. Workshop on Internet of Things Software Update (IoTSU)*, 2016.

[7] M. Ballano-Barcena and C. Wueest. Insecurity in the Internet of Things. *Symantec Security Response*, 2015.

[8] E.M. Stenberg. Key Considerations for Software Updates for Embedded Linux and IoT. *Linux Journal*, 2017(276), 2017.

[9] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proc. of the 2nd Conf. on Mobile Systems, Applications, and Services (MobiSys)*, 2004.

[10] B. Moran, H. Tschofenig, and H. Birkholz. Firmware Updates for Internet of Things Devices – An Information Model for Manifests. Internet-Draft draft-ietf-suit-information-model-01, 2018.

[11] E. Baccelli, J. Dörr, S. Kikuchi, F. Acosta-Padilla, K. Schleiser, and I. Thomas. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *Proc. of the IEEE Conference on Pervasive Computing and Communications (PerCom)*, 2018.

[12] C. Bormann, M. Ersue, and A. Keranen. RFC 7288: Terminology for Constrained-Node Networks. IETF, 2014.

[13] YUM: Yellow dog Updater, Modified. Website, http://linux.duke.edu/projects/yum.

[14] Debian APT Package Manager. Website, http://www.debian.org/doc/manuals/apt-howto/.

[15] S. Babic. swupdate – Software Update for Embedded Systems. Website, https://github.com/sbabic/swupdate, 2018.

[16] J. Appleseed. Mender: Over-The-Air Updater for Embedded Linux Devices. Website, https://www.mender.io/, 2018.

[17] RAUC: Robust Auto-Update Controller. Website, https://rauc.io, 2018.

[18] Balena.io: Container-based Software Updates for IoT. Website, https://www.balena.io/, 2018.

[19] M. Stolikj, P. Cuijpers, and J. Lukkien. Efficient Reprogramming of Wireless Sensor Networks using Incremental Updates. In *Proc. of the 9th IEEE Intl. Workshop on Sensor Networks and Systems for Pervasive Computing (PerSENS)*, 2013.

[20] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating Systems for Low-end Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(5):720–734, 2016.

[21] Z. Shelby, K. Hartke, and C. Bormann. RFC 7252: The Constrained Application Protocol (CoAP). IETF, 2014.

[22] TinyCrypt. Website, https://01.org/tinycrypt, 2018.

[23] Eclipse TinyDTLS. Website, https://projects.eclipse.org/projects/iot.tinydtls, 2018.

[24] V. Lakkundi and K. Singh. Lightweight DTLS Implementation in CoAP-based Internet of Things. In *Proc. of the 20th IEEE Intl. Conference on Advanced Computing and Communications (ADCOM)*, 2014.

[25] Nordic Semiconductor. nRF5 SDK: Device Firmware Update Process. Website, https://www.nordicsemi.com/DocLib/Content/SDK_Doc/nRF5_SDK/v15-2-0/lib_bootloader_dfu_process, 2018.

[26] Texas Instruments. Over the Air Download (OAD). Website, http://dev.ti.com/tirex/content/simplelink_cc2640r2_sdk_1_50_00_58/docs/blestack/ble_user_guide/html/oad-ble-stack-3.x/oad.html, 2018.

[27] RISE SICS. The Sparrow Application Layer and Tools. Website, https://github.com/sics-iot/sparrow, 2018.

[28] L. Tian. Lightweight M2M. Open Mobile Alliance Device Management Working Group, 2012.

[29] Apache. MCUmgr: A Management Library for 32-bit MCUs. Website, https://github.com/apache/mynewt-mcumgr.

[30] P. Levis et al. TinyOS: An Operating System for Sensor Networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[31] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. of the 1st Intl. Workshop on Embedded Networked Sensors (EmNetS)*, 2004.

[32] Contiki-NG: The OS for Next Generation IoT Devices. Website, https://github.com/contiki-ng/contiki-ng, 2018.

[33] J. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proc. of the 2nd ACM Intl. Conference on Embedded Networked Sensor Systems (SenSys)*, pages 81–94, 2004.

[34] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the IEEE Intl. Conf. on Computer Communications (INFOCOM) Workshops*, 2013.

[35] Zephyr OS. Website, https://www.zephyrproject.org, 2018.

[36] JUUL Labs. MCUboot: Secure boot for 32-bit Microcontrollers. Website, https://github.com/JuulLabs-OSS/mcuboot.

[37] A. Malhotra, I. Cohen, E. Brakke, and S. Goldberg. Attacking the Network Time Protocol. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[38] A. Malhotra, M. Van Gundy, M. Varia, H. Kennedy, J. Gardner, and S. Goldberg. The Security of NTP's Datagram Protocol. In *Proc. of the Conference on Financial Cryptography and Data Security*, 2017.

[39] Atmel Crypto Authentication. Website, https://www.microchip.com/design-centers/security-ics/cryptoauthentication.

[40] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable Key Compromise in Software Update Systems. In *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[41] D. Barrera and P. Van Oorschot. Secure software installation on smartphones. *IEEE Security & Privacy*, 9(3):42–48, 2011.

[42] H.A. Odat and S. Ganesan. Firmware over the air for automotive, Fotamotive. In *Proc. of the IEEE Intl. Conference on Electro/Information Technology (EIT)*, 2014.

[43] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos. Uptane: Securing Software Updates for Automobiles. In *Proc. of the 14th European Conference on Embedded Security in Cars*, 2016.

[44] S. Halder, A. Ghosal, and M. Conti. Secure OTA Software Updates in Connected Vehicles: A Survey. *arXiv preprint 1904.00685*, 2019.

[45] M. Steger, C.A. Boano, M. Karner, J. Hillebrand, W. Rom, and K. Römer. SecUp: Secure and Efficient Wireless Software Updates for Vehicles. In *Proc. of the Euromicro Digital System Design Conference (DSD)*, pages 628–636, 2016.

[46] M. Steger, C.A. Boano, T. Niedermayr, K. Römer, M. Karner, J. Hillebrand, and W. Rom. An Efficient and Secure Automotive Wireless Software Update Framework. *IEEE Transactions on Industrial Informatics*, 14(5):2181–2193, 2018.

[47] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of Things for Smart Cities. *Internet of Things Journal*, 1(1):22–32, 2014.

[48] H. Chandra, E. Anggadjaja, P. Wijaya, and E. Gunawan. Internet of Things: Over-the-Air (OTA) Firmware Update in Lightweight Mesh Network Protocol for Smart Urban Development. In *Proc. of the 22nd Asia-Pacific Conference on Communications (APCC)*, 2016.

[49] M. Solters. OTA for Contiki (CC2650 SoC). Website, http://marksolters.com/programming/2016/06/07/contiki-ota.html, 2018.

[50] NuttX. Website, http://nuttx.org/, 2018.

[51] Open Mobile Alliance. Lightweight Machine to Machine. Technical Specification OMA-TS-LightweightM2M-V1, 2013.

[52] Y. Zhong. Make Azure IoT hub SDK work on tiny devices. Website, https://azure.microsoft.com/en-us/blog/develop-for-constrained-devices.

[53] Microsoft Azure. Azure IoT C SDKs and Libraries. Website, https://github.com/Azure/azure-iot-sdk-c.