Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check

Koen Zandberg, Kaspar Schleiser, Francisco Acosta, Hannes Tschofenig, Emmanuel Baccelli

Secure Firmware Updates for Constrained IoT Devices using Open Standards: A Reality Check

Koen Zandberg
Inria & FU Berlin

Kaspar Schleiser
Inria & FU Berlin

Francisco Acosta
Inria

Hannes Tschofenig
Arm Ltd.

Emmanuel Baccelli
Inria

ABSTRACT

While IoT deployments multiply in a wide variety of verticals, most IoT devices lack a built-in secure firmware update mechanism. Without such a mechanism, however, critical security vulnerabilities cannot be fixed, and IoT devices can become a permanent liability, as demonstrated by recent large-scale attacks. In this paper, we survey open standards and open source libraries that provide useful building blocks for secure firmware updates for constrained IoT devices – by which we mean low-power, microcontroller-based devices such as networked sensors/actuators with a small amount of memory, among other constraints. We design and implement a prototype that leverages these building blocks and assess the security properties of this prototype. We present experimental results, including first experiments with SUIT, a new IETF standard for secure IoT firmware updates. We evaluate the performance of our implementation on a variety of commercial off-the-shelf constrained IoT devices. We conclude that it is possible to create a secure, standards-compliant firmware update solution that uses state-of-the-art security for IoT devices with less than 32kB of RAM and 128kB of flash memory.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**.

KEYWORDS

Internet of Things, IoT, Security, Software Update, Firmware Update, Open Standards, Constrained Device

1 INTRODUCTION

The increasing availability of low-cost hardware, new low-power radio technologies, and real-time operating systems specially designed for these embedded devices makes the Internet of Things (IoT) accessible to a broader range of developers. IoT devices are now used in many verticals, from logistics to precision farming, introducing new ways to optimize existing business processes and enabling novel use cases. IoT devices are also used in critical infrastructures where safety and security plays an even more important role.

However, while IoT devices are expected to have a major impact on our economy, they are also known for their weak security. The Mirai botnet [5], for example, demonstrated that large-scale DDoS attacks using compromised IoT devices threaten other communication infrastructures. It is equally alarming that many of these compromised IoT devices are not equipped with a firmware update mechanism and, therefore, remain *unpatched* to this day.

This highlights the need to design a firmware update mechanism into IoT devices at the beginning of the product development. Of course, if designed incorrectly, firmware updates can become attack vectors themselves. The Zigbee Worm [54], for example, triggered a chain reaction combining a series of malicious firmware updates and promiscuous wireless communications. The situation would be significantly improved if developers could use a standardized firmware update mechanism rather than having to design their own.

In this paper, therefore, we explore the options that developers have today, and we design a prototype that enables IoT firmware updates based on standardized building blocks.

We focus in particular on firmware update mechanisms that can work on constrained IoT devices. Such devices, as specified in RFC 7228 [19], use microcontrollers – for instance Arm Cortex-M – on which run real-time operating systems, such as RIOT, FreeRTOS, μC/OS, Contiki, mbed OS, among others [30]. Compared to machines that run full-blown operating systems, such as Linux, constrained IoT devices use a fraction of the power and are equipped with RAM and flash sizes in the kilobyte range. Constrained IoT devices cannot afford the energy drain of Wi-Fi, and thus connect to the network using low-power, wireless, link-layer technologies, such as Bluetooth Low-Energy, IEEE 802.15.4, LoRa, 3GPP Cellular IoT (NB-IoT), or through wired buses, such as BACnet.

The contributions of this paper are structured as follows:

(1) In Sections II-III, we survey available open standards and open source libraries, which provide useful generic building blocks that can be used to enable IoT firmware updates;
(2) In Section IV, we design and implement a prototype that leverages the building blocks we surveyed. This prototype enables secure firmware updates on a large variety of constrained IoT devices, while entirely avoiding proprietary mechanisms and code;
(3) In Section V, we measure and compare the performance of various crypto libraries that are relevant in this context;
(4) In Section VI, we assess the security properties of our prototype;
(5) In Section VII, we measure and compare the performance of several deployment configurations using our prototype, and provide the first experimental evaluation of the IETF SUIT specification;
(6) In Section VIII, we discuss the limitations of our prototype. We conclude that, as we have shown, it is possible today to create a generic, secure firmware update mechanism that

complies with open standards, and we provide recommendations for future work.

2 PRIOR WORK ON SOFTWARE UPDATES FOR CONSTRAINED IoT DEVICES

An IoT firmware update solution is a special case of software update, and consists of three areas of work [23], namely: (a) embedded software design on low-end IoT devices, (b) backend framework, and (c) network transport of the firmware towards the IoT devices.

Embedded software design on low-end IoT devices. The software on an IoT device has to be prepared to support a firmware update mechanism. The device needs a *bootloader*, the logic that is executed first when the device boots and determines which firmware it launches. Sometimes devices are equipped with multiple bootloaders; for example, a stage 1 bootloader in the ROM and a stage 2 bootloader that can be updated. The reason for such designs is security-related because updating a bootloader can lead to a bricked device. Whenever a bootloader is present on a device, the memory layout of the hardware has to be considered, and exception handlers[1] must be repositioned.

The typical firmware update procedure is fairly simple: a developer recompiles the code and generates an entirely new firmware image, which is then distributed to the device. The flash memory of the IoT device is split into memory regions (slots) containing (i) the bootloader and (ii) firmware images (with some metadata). The new firmware is stored into one of the available slots. The IoT device is then reset so that the bootloader can boot the new firmware image [10]. This approach is used, for example, by MCUboot [2] and ESPer [27].

Other considerations can lead to different designs. For instance, one may consider the granularity of the software update, or the amount of data that needs to be transmitted for an update. Certain approaches enable partial update via dynamic loading of binary modules [26, 55], while others use differential binary patching [33]. Yet another technique is binary compression [60]. Approaches using component-based programming [64, 65] aim to simplify dynamic modification and reconfigurability of the system on constrained IoT devices by enforcing black-box-style interactions between system modules. Partial updates of software can also use scripts instead of binaries [15], whereby pieces of interpreted language (for example, Javascript) are updatable on devices. Yet another technique uses miniature virtual machines, such as Mate [39] or ReLog [64].

Despite the above-listed research, the typical approach used for IoT software updates is to replace the full firmware image at once. The advantage of updating the full firmware is in the simplicity of this approach.

Backend framework. The second aspect of IoT firmware updates concerns the backend framework and securing the supply chain of IoT software. The Internet Engineering Task Force (IETF) Software Updates for Internet of Things (SUIT) working group specifies a simple back-end architecture [43] for IoT firmware updates. In addition to authentication and integrity protection, even when updates are stored on untrusted repositories, the SUIT specifications enable encrypting the firmware image, to protect against attacks based on reverse engineering. SUIT followed previous work such as FOSE [?] which proposed firmware encryption and signing using JSON and JOSE. The Update Framework (TUF) [3] and Uptane [37], designed for use in connected cars, aim to ensure the security of a software update system, even against attackers who compromise the repository or signing keys. ASSURED [14] builds on TUF to improve support for constrained IoT devices by leveraging a trusted intermediate controller between the update repository and IoT device. CHAINIAC [46] is another approach that uses a blockchain-like mechanism to attest to the history of prior updates, even without central authority.

Network transport. The third aspect of IoT firmware updates concerns the dissemination of software through the network. The variety of approaches to this topic, as presented in recently published literature, includes protocols that optimize the dissemination of updates through multiple paths in a multi-hop, low-power wireless network [31]; updating network stack modules to reconfigure the network on the fly [65]; and using the Message Queuing Telemetry Transport (MQTT) protocol to disseminate software updates to a fleet of IoT devices [27]. 6LoWPAN protocols [58] enable end-to-end IP connectivity from constrained IoT devices to anywhere on the (IPv6) internet. The IETF Trusted Execution Environment Provisioning (TEEP) working group [32] is standardizing a transport mechanism to update trusted applications running in trusted execution environments (TEEs), such as Arm TrustZone and Intel SGX.

3 OPEN STANDARDS FOR SECURE CONSTRAINED IoT FIRMWARE UPDATES

Over the last few years, the technical community has been working on open standards [36] that can be combined to facilitate IoT firmware updates. These open standards fall into the following categories:

- Cryptographic algorithms;
- Firmware metadata;
- Protocols for transferring updates over the network;
- IoT device management protocols.

We also have to consider IoT operating systems for use in our prototype.

Cryptographic algorithms. The use of state-of-the-art cryptographic algorithms is necessary to guarantee the security of firmware updates. For many years, the impression was that algorithms used on the wider Internet could not be used on constrained IoT devices. This turned out to be incorrect; however, optimization and selection of different algorithms is necessary. For public key cryptography, Elliptic Curve Cryptography (ECC) is typically used because of the smaller key size (compared to RSA). The National Institute of Standards and Technology (NIST) standardized the Elliptic Curve Digital Signature Algorithm (ECDSA) for use with the P256r1 curve [48], which became popular in the industry. With ed25519 [35], another signature algorithm, based on a different

[1]Exception handlers in the Arm architecture can be compared to the interrupt vector table in the x86 architecture.

curve, was standardized. New standardization efforts are in progress to evaluate algorithms for the post-quantum crypto area [47].

Firmware metadata. The IETF SUIT working group is currently standardizing a format for describing firmware updates. The SUIT group defines a so-called *manifest*, which provides (1) information about the firmware required to update the device, and (2) a security wrapper to protect the metadata end-to-end.

Taking TUF/Uptane [37] as a reference, for instance, the SUIT manifest format could provide Uptane-compliant (custom) metadata about firmware images. (The TUF standards neither target interoperability, nor specify concrete metadata formating, contrary to the SUIT standards.)

The SUIT specifications include an architecture document [43], an information model description [44], and a proposal for a manifest specification [45].

To achieve its goals, SUIT builds upon a number of other open standards that provide generic building blocks. In particular, the Concise Binary Object Representation (CBOR) [20] specification is used as a data format for serialization. CBOR is a schema-less format optimized for a small message size using a binary encoding. Furthermore, the CBOR Object Signing and Encryption (COSE) [56] specification is used to cryptographically secure data serialized with CBOR. COSE defines a variety of structures, among them the *sign* structure, which specifies how to protect a payload against tampering by using a cryptographic signature.

Standards for IoT firmware transport. A number of protocols provide specifications for transferring a firmware update over the network. Basic transport schemes enable a so-called Device Firmware Update (DFU) over a specific low-power Media Access Control (MAC) layer technology (such as Bluetooth), or over a specific bus technology (such as USB). On the other hand, to transport firmware over several hops, or over heterogeneous low-power networks, the IETF suite of protocols standardized a network stack combining Constrained Application Protocol (CoAP) over UDP [57] and CoAP over TCP/TLS [58]. CoAP offers features equivalent to HTTP but tailored to constrained IoT devices. The 6LoWPAN specification was designed to offer an adaptation layer for networks that cannot directly use IPv6. To provide communication security, DTLS and TLS profiles [61] were standardized for use in IoT deployments.

Standards for remote IoT device management. The most prominent open standard for IoT device management is the Lightweight Machine-to-Machine (LwM2M) protocol [49–51] developed by OMA SpecWorks[2]. To transfer data, LwM2M v1.1 uses CoAP, which can be secured with DTLS [61] . The LwM2M specifications define a simple data model and several RESTful interfaces for remote management of IoT devices. The interfaces enable devices to register to a server, provide information updates, and obtain keying material. A large number of objects and resources have already been standardized to support commonly used sensors, actuators, and other resources. Among the standardized objects is the *firmware update* object.

A more recent design is the CoAP Management Interface (CoMI) [63], which is standardized by the IETF. CoMI uses CoAP and a data model based on the YANG modeling language, and aims to reuse existing SNMP-defined objects and resources. CoMI is still in development, and a firmware update mechanism has not yet been defined; we do, however, expect that such an extension will be defined in the future.

The Open Connectivity Foundation (OCF[3]) standardizes an IoT device management protocol operating on top of CoAP and TLS/DTLS for communication, similarly to LwM2M. The OCF defines a data model with RESTful API Modeling Language (RAML) as the data modeling language. While initially targeting bigger IoT devices in smart home environments, the OCF is now also considering other industry verticals.

Earlier work on device management for IoT devices use remote procedure calls instead of a RESTful design. For instance TR 69 [21], also known as the CPE WAN Management Protocol (CWMP) developed by the Broadband Forum[4] offers firmware update functionality on higher-end IoT devices, such as Internet-connected printers. The successor of TR 69, called User Services Platform (USP) [22], was recently released by the Broadband Forum.

De facto standard IoT operating systems. Off-the-shelf open source operating systems, such as Linux, cannot be used on low-end IoT devices, which lack the necessary hardware resources. Unfortunately, the increasing complexity of Internet-connected devices requires a fairly complex protocol stack, which includes IPv6, UDP, DTLS, and CoAP.

This situation has led to the development of IoT operating systems, including many open source IoT operating systems [30], such as RIOT [16], Zephyr [4], Mbed OS [12], MyNewt or Tock [40]. Popular commercial operating systems in this category include μC/OS [38] and FreeRTOS [11].

4 PROTOTYPE DESIGN

In this section, we describe a prototype we designed to implement the functionality required by the scenario described in 4.1, below. A link to the source code is provided in the References section at the end of this article [1].

4.1 Scenario Setup

Prior work [43] outlines requirements for firmware updates of IoT devices, and lists various common deployment scenarios. In this paper, we consider the scenario shown in Fig. 1 for further investigation. In this scenario, an IoT device is connected through a low-power wireless network to a device management server, which runs on the internet.

Over the lifetime of this IoT device, an authorized IoT software maintainer should be able to:

(1) Produce firmware updates that are integrity-protected and authenticated;
(2) Trigger the device to fetch (via push or pull) and verify the integrity and authenticity of a firmware image, and then reboot;

[2]OMA SpecWorks is the result of a merger between the Open Mobile Alliance (OMA) and the IP Smart Object (IPSO) Alliance.

[3]OCF is the result of a merger between the UPnP Forum, the Open Interconnect Consortium (OIC), and the AllJoyn Alliance.

[4]The Broadband Forum was formerly known as the DSL Forum.

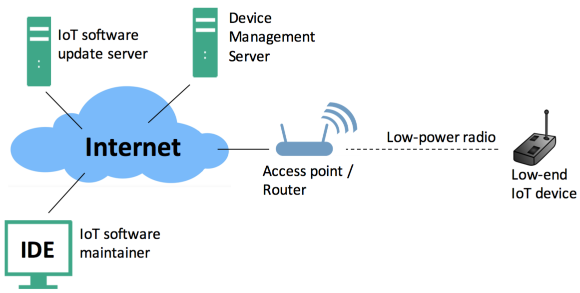

Figure 1: IoT Firmware Update Prototyping Scenario.

(3) Delegate authorization to another maintainer, in case of new ownership or change of contracts (we use the same technique to switch trust anchor when it expires or has to be revoked);
(4) Reconfigure the device so that cryptographic algorithms can be upgraded, if needed.

There are several aspects we do not explore in the prototype we aim for:

- We only consider the case where the entire firmware is replaced. We do not consider differential updates.
- We focus on the use of asymmetric cryptography for digital signatures, although a symmetric key solution is also possible.
- We do not make use of firmware encryption.
- We avoid proprietary protocols, focus only on open source software, and aim for simplicity; therefore, we do not explore optimization potential. Our results should therefore be interpreted as representing the ”lower bar”.

We designed this prototype such that multiple configurations are possible – for example, to switch crypto algorithms, crypto libraries, and network stacks – and the same code can be executed on IoT hardware from different vendors. This provides us with a good basis for comparing different features.

4.2 Components and Functional Overview

Based on our survey of applicable open standards in Section 3, we utilize the following building blocks:

- The firmware metadata format based on the IETF SUIT manifest.
- The 6LoWPAN, IPv6, and CoAP transport stack.
- The LwM2M IoT device management solution.
- Digital signature algorithms based on ed25519 and ECDSA/P256r1.

We selected the RIOT [16] operating system for this prototype, but the results can easily be transferred to other real-time operating systems.

The remainder of this section provides a functional overview of the prototype.

IoT device commissioning. From the embedded software point of view, the prototype we built is based on the design shown in Fig. 2, articulating: (i) a minimalistic bootloader, (ii) two firmware image slots in flash memory, each prefixed with space for their respective metadata, and (iii) a basic firmware update module, also implemented on top of RIOT, integrated into each firmware image, as detailed below.

We enhanced the RIOT build system to enable a maintainer to simultaneously build and flash (through the serial or USB port) the bootloader and the initial firmware in the first slot. The initial firmware includes a software module for firmware updates, configured with the necessary trust anchor of the maintainer.

Trust anchor. Our model is based on a single trust anchor, namely of the authorized maintainer. This trust anchor is used to verify the authenticity of the signed firmware image. If an attacker manages to trick the maintainer into handing out the private key associated with the trust anchor, the attacker can load malicious firmware images onto the IoT device. An attacker could make the compromised maintainer sign malicious firmware images. Alternatively, the compromised maintainer could relinquish authorization to the attacker. There is no mitigation when the only trust anchor used is compromised. In this prototype, therefore, we rely on the maintainers' ability to keep their private keys secure. Extensions using a public key infrastructure, potentially with a hierarchy of keys, is possible but out of scope for this paper.

Producing and uploading an authorized firmware update. We enhanced the build system so that a maintainer – a software developer – can simultaneously build a new firmware image and produce the corresponding metadata, signed with the private key of the maintainer. The firmware and signed metadata can then be uploaded to the IoT software update server, using an HTTP-based API. The update server is a web server, which can speak both HTTP and CoAP. It interfaces with the maintainer of the firmware and with the IoT device.

Firmware update module. The firmware update module's main tasks are to retrieve the firmware image and manifest from the update server, to parse and verify the manifests, and to store the firmware image on flash memory.

The module implements the required buffering between the network packet size and the device flash page size. When a flash page buffer is full, the module writes the buffer to the next flash page in the (non-active) firmware image slot. After the entire firmware image has been written to flash, the module computes a hash and checks that this hash is identical to the hash announced in the transferred firmware's metadata. The received metadata is cryptographically verified with the help of the trust anchor (the public key stored on the device). If the digital signature is verified, and if other security checks pass (for example, the firmware sequence number is confirmed to be newer), the module also writes the metadata to the flash (otherwise, the metadata is blanked) and launches a reboot. The bootloader then reads the metadata from the two available firmware slots and chooses to boot the newest valid firmware, based on the metadata. Note that, due to blanked metadata, an interruption (e.g. due to power loss) cannot cause the system to boot of an invalid, corrupted or incompletely received image.

Scheduling firmware updates. Using the firmware update module, updates can be (i) either triggered periodically or on demand,

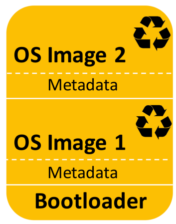

Figure 2: Embedded Flash Memory Layout.

	Firmware Update	IPv6 Support	Standardized Manifest	Device Mgmt
Baseline	×	✓	×	×
Basic-OTA	✓	×	×	×
IPv6-OTA	✓	✓	×	×
SUIT-OTA	✓	✓	✓	×
LwM2M-OTA	✓	✓	✓	✓

Table 1: Analyzed Configurations.

(ii) pushed to the device or pulled from the device [**?**], so as to fit other operational constraints. On the device we use the real-time, preemptive multi-threading capabilities of RIOT, such that the system is not blocked by the computational-intensive task of digital signature verification. In practice, signature validation runs in a separate thread, with low priority, enabling other threads with top priority to execute as needed. However, note that we do not target more advanced fine tuning for the schedule of firmware updates (e.g. to guarantee the continuity of some service provided by the device, or to optimize network load). Instead, we focus primarily on the fundamental embedded system characteristics and constraints imposed by standard-compliant firmware update onboard constrained IoT devices.

Lifecycle management. By changing the trust anchor stored in the next firmware's update module, authorization to update the firmware can be delegated to another maintainer, who can take over the production and the roll out of authorized updates.

Crypto agility is straightforward because the update module in the new firmware image can implement and use upgraded cryptographic primitives. This flexibility is provided because we implement the cryptographic primitives in the software.

Key roll-over is also made possible with the ability to update the trust anchor.

4.3 Configurability of the Prototype

The prototype we designed can be configured in multiple ways, as summarized in Table 1.

We created the following configurations:

Baseline. The Baseline configuration covers a typical sensor scenario, and is introduced here only as a reference, to evaluate the relative cost of over-the-air (OTA) firmware updates. Therefore, this configuration does not provide firmware update functionality. The Baseline configuration uses 6LoWPAN over IEEE 802.15.4 as a network stack. A CoAP server is installed on the IoT device to respond to requests for sensor data and to actions that trigger an actuator.

Basic-OTA. This configuration enables over-the-air firmware updates pushed directly from the update server to the IoT device, over the MAC layer, without a standard network layer. Therefore, this Basic-OTA configuration requires that the IoT device and the update server can communicate directly over the MAC layer (in other words, they have to be on the same local network/bus). The Basic-OTA configuration uses minimalistic firmware metadata (in a proprietary format), namely:

- A sequence number.
- The firmware start address and size.
- A digest of the firmware image.
- A digital signature of the metadata.

IPv6-OTA. This configuration enables the Basic-OTA configuration by using an IPv6-compliant network stack. The IPv6 network layer implementation is provided by the RIOT Generic (GNRC) network stack. CoAP blockwise transfer (block1) is used because UDP limits the size of the firmware image to be transferred to 65,507 bytes and, more importantly, we want to avoid the inefficiency caused by IP fragmentation.

SUIT-OTA. This configuration implements firmware updates following the IETF SUIT manifest [45]. Compared to IPv6-OTA and Basic-OTA, SUIT-compliant firmware metadata offers more features and additional security guarantees (see Section 6).

The SUIT manifests used in our prototype contain the following information:

- The firmware version number.
- An 8-byte nonce.
- A sequence number (whereby we use the current time).
- A single condition: limiting the validity of the manifest to our device.
- The format of the firmware.
- The size of firmware.
- A storage identifier.
- A single URI to allow the device to download the firmware.
- A SHA256 digest.
- A digital signature on the manifest.

Upon receiving a manifest, the IoT device checks the signature, and, if verified correctly, pulls the firmware from the URI indicated in the SUIT manifest. To pull the firmware image, we again use CoAP blockwise transfer (block2). It would be possible to attach the firmware to the manifest, but using this two-step approach gives us extra flexibility.

LwM2M-OTA. This configuration adds support for LwM2M v1.0 (without the use of the bootstrapping functionality). The device registers to a LwM2M server and provides the necessary API endpoints complying with the LwM2M specification and the core objects, such as the LwM2M Device and the LwM2M Firmware Update objects. The firmware is updated by pushing a SUIT manifest to the Package resource found in the LwM2M Firmware Update object followed by the workflow corresponding to the SUIT-OTA configuration.

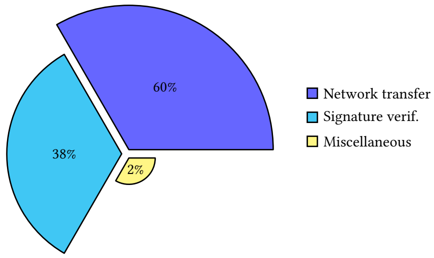

Figure 3: Time spent per subtask in a firmware update.

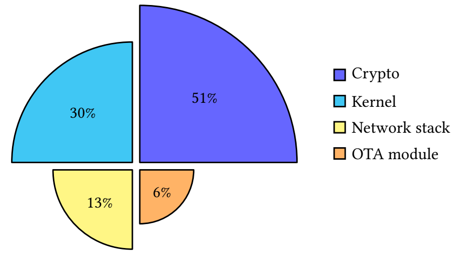

Figure 4: Flash memory budget per system component (Basic-OTA configuration, 35kB flash in total).

In the analyzed configurations above, we have not used TLS/DTLS between the IoT device and the update server (or device management server for LwM2M). Implementing TLS/DTLS is certainly useful when considering the larger device management functionality in addition to the firmware update. An analysis of IoT device management functionality is, however, outside the scope of this paper.

4.4 Relative Impact of Crypto

In such a system, cryptography significantly impacts memory and power budgets. To get an idea of how much, we measured the relative memory budget and time spent due to crypto for the Basic-OTA configuration of our prototype (using the HACL crypto library [66]). First, we observe in Fig. 3 that, compared to the time (and thus energy) needed for signature verification and network transport, negligible time (less than 2%) is spent on network packet handling and parsing, as well as on firmware metadata parsing and validation (excluding signature verification). Note that this remains true with other configurations of our prototype as well, using a more elaborate network stack (CoAP) or more elaborate metadata (SUIT). Next, we observe in Fig. 4 that crypto represents 50% of the memory budget. Going back to Fig. 3, it seems at first sight that time spent during a firmware update is dominated by network transfer (60%) then signature verification (38%) as shown in Fig. 3. However, we observe that, since half of the firmware image size is contributed by cryptographic functions, this means 30% of the time is spent on transferring updated crypto over the network (half the network transfer time). In effect, we conclude that handling cryptography dominates, accounting in fact for 68% of the total time spent on the firmware update process. We conclude that choosing an appropriate cryptographic algorithms and library, offering a good compromise on size/speed, is crucial. In the following section, we discuss this topic in greater detail.

5 CRYPTOGRAPHIC LIBRARIES

Our prototype makes use of cryptographic primitives for verifying digital signatures. Thus, a crucial parameter is the choice of a signature algorithm, and its implementation.

In the context of this paper, we considered algorithms that provide a cryptographic strength of roughly 128-bit. Concretely, we considered mainly signatures based on ed25519 and signatures based on the NIST P256r1 curve.

Based on the choice of algorithms, a particular implementation must then be chosen. We briefly survey a number of relevant libraries, many of which are highly configurable, and evaluate them in Section 7.2.

subsubsectionLibraries that provide ed25519 signatures

HACL* [66] is a cryptographic library that is written in the F* programming language and is compiled to C. The goal of HACL* is to have a verified cryptographic library. In this paper, we evaluate the version from the HACL-C repository [29] (commit d65ee4).

TweetNaCl [17] is a small portable and auditable C library implementing all 25 functions of NaCl. It is simple to use and does not offer configurable options. It provides one combination of cryptographic primitives selected for high security. In this paper, we evaluate version TweetNaCl-2014-04-27.

μNaCl [25] is a patch on top of TweetNaCl. μNaCl provides assembly code optimized for some microcontroller architectures, and aims to significantly speed up the x25519 key exchange. In this paper, we evaluate version μNaCl-2015-08-13.

C25519 [6] is a library that implements ed25519 for embedded implementations. The memory consumption and code size of C25519 is small. In this paper, we evaluate C25519-2017-10-05.

Monocypher [62] is a small auditable library implementing the ed25519 signature scheme among other cryptographic primitives. It aims to keep the code base small while not sacrificing too much speed. In this paper, we evaluate version Monocypher-2.0.5.

WolfSSL [9] is an embedded TLS/DTLS library. The wolfCrypt module is used here to measure raw crypto performance. In this paper, we evaluate a version of WolfSSL adapted for integration in the RIOT operating system (based on commit 412eecd).

5.0.1 *Libraries that provide P256r1 signatures.*

TinyCrypt [8] is a cryptographic library that provides signatures based on the NIST P256r1 curve [41]. The design goals of TinyCrypt are to minimize the code size and cryptographic dependencies. In this paper, we evaluate version TinyCrypt-0.2.8.

Mbed TLS [13] is an embedded TLS/DTLS library. In this paper, we evaluate version mbedTLS-2.12.0.

Note: Although WolfSSL and Mbed TLS are TLS/DTLS stacks, we only use the implementations of the cryptographic algorithms in the prototype; not in the TLS/DTLS protocol itself.

5.0.2 *Other digital signature libraries.*

There are also alternative digital signature schemes that provide

Scheme	Library	Flash	Stack	Speed	
				M0+	M4
ed25519	HACL*	★★★★★	★★★★	★★★★★	★★★★
	TweetNaCl	★★	★★★★★	★★★★★	★★★★
	uNaCl	★★	★★★★★	★★★★★	★★★★
	C25519	★★	★★	★★★	★★★★★
	Monocypher	★★★★	★★★★★	★	★
	WolfSSL	★★	★★	★★★	★★★★★
P256r1	TinyCrypt	★★	★	★	★
	Mbed TLS	★★★★★	★	★	★★★
Other	qDSA	★★★★★	★	★	★★★★
	Libhydrogen	★	★	★	★

Table 2: Crypto library performance summary (Fewer stars ★ is better.)

	IPv6-OTA	SUIT-OTA	LwM2M-OTA
A. Tampered firmware	✓	✓	✓
B. Firmware replay	✓	✓	✓
C. Offline device	×	×	✓
D. Firmware mismatch	×	✓	✓
E. Wrong memory location	✓	✓	✓
F. Unexpected precursor	×	✓	✓
G. Reverse engineering	×	×	×
H. Resource exhaustion	×	×	✓

Table 3: Security Assessment Summary.

the same (128-bit) level of security. For instance, recent work on hyperelliptic curves includes **qDSA** [53], a signature scheme that yields fast signatures and verifications for constrained IoT devices. An implementation of qDSA is available as a RIOT package [52], which provides a reference C implementation and assembly code optimizations for both AVR and Arm Cortex M0 microcontrollers.

Other work is based on variants with the Gimli permutation [18]. For instance, **libhydrogen** [7] implements such a signature scheme and enables fast, small cryptographic signatures.

5.0.3 Summary.
The performance of the digital signature libraries we surveyed above are evaluated and compared in detail in Section 7.2. A high-level summary of the (significant) differences we observed comparing these libraries in terms of speed and memory requirements in RAM and Flash can be found in Table 2. Based on this summary, we chose to configure and evaluate our prototype firmware update using ed25519 signatures provided by the C25519 library, which offers a good speed/size compromise.

6 SECURITY ASSESSMENT

Typical threats against a firmware update solution are discussed in the SUIT information model [44] and can be categorized into (i) privilege escalation, (ii) device malfunction, (iii) resource exhaustion, (iv) reverse engineering, and (v) social engineering.

Based on these threats, we assess and compare the security of our prototype in the IPv6-OTA, SUIT-OTA, and LwM2M-OTA configurations, which are defined in Section 4.3. The summary of our assessment is shown in Table 3.

6.1 Tampered firmware

An attacker may try to update the IoT device with a modified and intentionally flawed firmware image. To counter this threat, the IPv6-OTA, SUIT-OTA, and LwM2M-OTA configurations use digital signatures to ensure integrity of both the firmware and its metadata. Additionally, the device can verify that an authorized maintainer signed the firmware image.

6.2 Firmware replay

An attacker may try to replay a valid, but old (known-to-be-flawed) firmware. This threat is mitigated by using a sequence number. All three configurations use a sequence number, which is increased with every new firmware update.

6.3 Offline device attack

An attacker may cut communication between the IoT device and the update server for an extended period of time. Then, he or she may try to update the IoT device with a (known-to-be-flawed) firmware image, which has in the meanwhile been deprecated. IPv6-OTA does not provide any mitigation against this threat.

Following the SUIT specification, a *best-before* timestamp can be used to expire an update. However, this requires the IoT device to have an approximate knowledge of the current date/time, which may not be available on constrained IoT devices. Therefore, our SUIT-OTA configuration does not mitigate this threat either. Only the LwM2M-OTA configuration may protect against this attack since LwM2M offers an easy way to provision the device with time information.

6.4 Firmware mismatch

An attacker may try replaying a firmware update that is authentic, but for an incompatible device. While IPv6-OTA does not provide mitigation against this threat, the SUIT-OTA and the LwM2M-OTA configurations include device-specific conditions, which can be verified before installing a firmware image, thereby preventing the device from using an incompatible firmware image.

6.5 Flash memory location mismatch

An attacker may attempt to trick the IoT device into flashing the new firmware to the wrong location in memory. To mitigate this attack, IPv6-OTA, SUIT-OTA, and LwM2M-OTA specify the intended memory location of the firmware update.

6.6 Unexpected precursor image

An attacker may try to exploit a vulnerability that results from a mismatch between previously installed software and the new firmware. While IPv6-OTA does not mitigate this threat, SUIT-OTA and LwM2M-OTA enable specifying the precursor software that must be installed before the update can be applied (enabling modular/incremental updates).

6.7 Reverse engineering

The firmware image in transmission can be captured by an attacker for vulnerability analysis. Neither the IPv6-OTA configuration nor our SUIT-OTA configuration protect against eavesdropping end-to-end (from the maintainer to the IoT device). Note that the SUIT specification also defines the ability to encrypt the firmware image; however, our prototype does not make use of this feature. The use of (D)TLS in the SUIT-OTA or LwM2M-OTA configurations can also protect the firmware image against eavesdropping in-flight, while transmitted over the network, but doesn't offer end-to-end security without the extra protection offered by using SUIT.

6.8 Resource exhaustion

Receiving, verifying, and storing a new firmware is an operation that typically uses up a significant amount of resources on a constrained IoT device. As discussed in Section 7.2, signature verification can take several seconds. By repeatedly attempting fraudulent firmware updates, an attacker may deplete the device's battery or, more generally, make it unavailable for long periods of time. For example, an attacker who manages to transmit valid manifests without a valid signature to an IoT device at regular intervals can drain the battery.

The IPv6-OTA configuration does not mitigate this threat, but the SUIT-OTA configuration lowers the impact by verifying the manifest before downloading the firmware image. However, an attacker could still push invalid manifests at any rate, causing the IoT device to perform signature verifications. Using LwM2M, an additional layer of defense can be added by only processing manifest that are conveyed via the device management infrastructure. In this way, the IoT device trusts the LwM2M server to only forward manifests that pass the following security checks:

- The URL in the manifest points to a firmware update server under the control of the LwM2M infrastructure.
- The manifest signature has been verified correctly.
- Other conditions in the manifest (such as the *best-before* timestamp) have been processed successfully.

If the device management server is compromised, the security characteristics of the LwM2M-OTA configuration fall back to those of the SUIT-OTA configuration.

7 EXPERIMENTAL PERFORMANCE EVALUATION

The quantitative analysis of our prototype is split into two parts. First, we analyze the firmware update solution. Second, we evaluate the performance of various crypto libraries for use with our firmware update solution.

IoT hardware. For the performance analysis, we use commercially available hardware based on Arm Cortex M microcontrollers. We use the following hardware from three different vendors:

- Atmel SAMR21, which features a Cortex M0+ MCU with 32 kB of RAM and 256 kB of flash.
- STM32F103REY, which features a Cortex M3 MCU with 64 kB of RAM and 512 kB of flash.
- Nordic nrf52840, which features a Cortex M4 with 256 kB of RAM and 1 MB of flash.

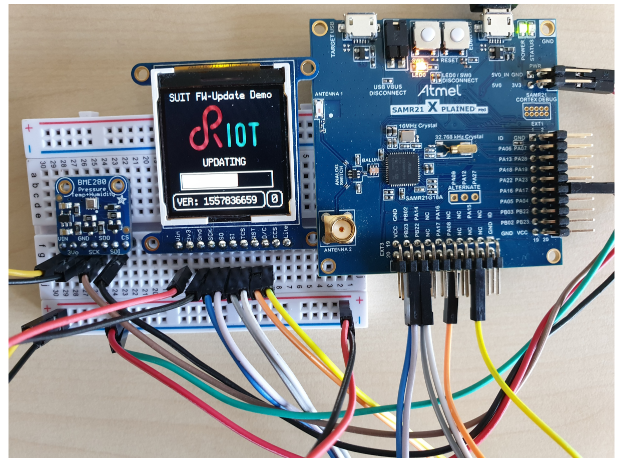

Figure 5: Hardware Prototype (using SAMR21).

The STM32F103REY and the nrf52840 are clocked at 64Mhz, while the SAMR21 runs at 48Mhz. Fig. 5 shows a hardware prototype we used (version using the Atmel SAMR21). In the following measurements, the code is compiled using GCC 7.2.0 for Arm optimized for code size. To retain generality, we focus only on the software necessary to enable secure firmware updates over the network, i.e. we ignore application-specific code and memory such as the driver for the LCD screen shown in Fig. 5.

Metrics. To evaluate cost in our comparative evaluation, we use both (i) memory measurements (RAM and flash size) and (ii) CPU performance measurements. These metrics are decisive in terms of hardware costs and in terms of energy costs [24].

On the one hand, a slower CPU speed and larger RAM size increase energy consumption. On the other hand, a faster CPU with more RAM and flash memory typically increases the price of the MCU[5]. In practice, memory sizes available for off-the-shelf microcontrollers are typically of size 2^n (for example, 32 kB RAM and 256 kB flash, or 16 kB RAM and 128 kB flash). Therefore, hardware design constraints are substantially impacted when such thresholds are crossed.

7.1 Evaluating the Cost of the OTA Update Functionality

To evaluate the cost of the firmware update functionality, we measured and compared the RAM and flash memory overhead incurred by this functionality in our prototype for the various configurations we defined in Section 4.3. The flash memory footprints (total

[5]The price of an MCU is determined by many factors, including economies of scale. Therefore, it may even be the case that an MCU with a better hardware layout is cheaper than a more constrained MCU.

Component	Bootloader	Baseline	Basic-OTA	IPv6-OTA	SUIT-OTA	LwM2M-OTA
Core	2 760	13 976	10 913	13 241	14 388	14 175
Network	0	26 892	2 732	26 892	27 230	27 208
CoAP	0	1 876	1 910	1 910	2 286	2 676
Crypto	0	308	5 798	5 886	6 472	6 472
COSE + CBOR	0	0	0	0	3 181	3 181
SUIT	0	0	0	0	1 575	1 551
OTA	0	0	2 007	2 007	3 998	3 475
LwM2M	0	0	0	0	0	2 166
Sub-total per image	2 760	43 052	23 360	49 936	59 130	60 904
Total flash footprint	2 760	43 052	49 544	102 696	121 084	124 632

Table 4: Flash requirements (in bytes) per component and configuration, on Cortex M0+.

Component	Bootloader	Baseline	Basic-OTA	IPv6-OTA	SUIT-OTA	LwM2M-OTA
Core	800	2 410	1 317	2 410	3 914	3 919
Network	0	11 010	7 224	11 010	11 010	11 026
CoAP	0	1 536	2 560	2 560	1 024	1 024
Crypto	0	28	28	28	60	60
COSE + CBOR	0	0	0	0	512	512
SUIT	0	0	0	0	296	272
OTA	0	0	632	632	2 984	3 000
LwM2M	0	0	0	0	0	1 487
Total	800	14 984	11 760	16 640	19 800	21 300

Table 5: RAM requirements (bytes of statically allocated stack) per component and configuration, on Cortex M0+.

and broken down per component) are shown in Table 4, while Table 5 shows the RAM requirements calculated for the stack[6] measured on an Atmel SAMR21 (using a Cortex M0+, the most constrained MCU we used in our experiments). In these two tables, we also list the bootloader as a separate item because it is present on the device alongside the firmware images, as shown in Table 2.

We distinguish between different components in the system as follows:

- The *core* component combines the minimal basic operating system functionality, including drivers. The newlib-nano library is also included.
- The *crypto* component includes cryptographic algorithms, such as digest algorithms, the digital signature algorithm, the ECC and bignum library, and the pseudo random number generators.
- The *network* component includes the protocol stack from the radio driver up to the transport layer protocol UDP.
- The modules that enable a firmware update to be received and stored in flash memory are combined in the *OTA* component.
- *CoAP* refers to the CoAP protocol stack.
- *COSE+CBOR* contains the libraries for COSE parsing and CBOR parsing.
- *SUIT* relates to the code parsing a SUIT manifest.
- Finally, *LwM2M* contains the code for device registration, and functionality required for the LwM2M protocol to perform firmware updates (particularly the LwM2M Device and Firmware Update objects).

[6]We measure the RAM utilization using static analysis; that is, based on the compiler-generated call graphs. This technique is simple and a good approximation. It does, however, produce inaccuracy when the code contains assembly language and function pointers. Dynamic analysis, on the other hand, is not perfect either because it does not easily indicate the maximum stack size. Note that memory allocations on the heap are not considered in our measurement.

7.1.1 *The Cost of OTA.*

The cost of basic OTA functionality can be measured by comparing the memory requirements of the Baseline configuration with that of the IPv6-OTA configuration. On a per-image basis, the flash overhead comes from the need for additional modules to perform necessary crypto (5 kB) and to handle OTA (2 kB). However, the prototype needs two image slots with metadata and a bootloader. We are, therefore, comparing the Baseline flash footprint against twice the flash footprint of IPv6-OTA added with the bootloader footprint (see Table 4). In total, the relative overhead in flash memory footprint is 137% (59 kB more). Note that this overhead means that the flash memory budget crosses over from below 64 kB to below 128 kB. The largest part of the overhead comes from the doubled image slots. The footprint of the rest (bootloader and metadata) is small: approximately 3 kB of flash for the bootloader and a single flash page for the metadata of each image. Due to flash memory alignment constraints, the size of the metadata is effectively rounded up to a full flash page stored in the memory of the IoT device.

On the other hand, RAM requirements increase by 3 kB, and could most probably be kept under the 16 kB threshold with additional optimization. The memory footprint overhead can be reduced if standard-compliance is dropped at the network level. For instance, the Basic-OTA functionality stays below 64 kB flash.

7.1.2 The Cost of Standard-Compliance for OTA.

The use of standards-compliant specifications, such as 6LoWPAN, SUIT, and LwM2M, increases the memory footprint due to the extra functionality provided; for example, serialization, metadata processing, and object handling. This is expected.

However, we observe that the relative overhead per image, compared to the Baseline scenario, is small. This is because a lot of features are reused in the network module. Furthermore, it is not unlikely that, OTA functionality aside, application code already leverages CBOR, COSE, and other crypto functionality. In such cases, the extra memory overhead per image falls to approximately 10%. This type of software reuse is a clear advantage of using standard building blocks.

Compared to the 124 bytes of metadata transferred over the network with the Basic-OTA configuration, 226 bytes of metadata need to be transferred with the SUIT-OTA configuration (counting full COSE data).

Due to the flash memory alignment constraints on the IoT device, this overhead has no effect on the flash memory footprint because 226 bytes typically fit on a single flash page (for example, 256 bytes fit on a single flash page on the SAMR21, the most constrained MCU we used in our measurements).

Finally, we observe that none of the configurations we experimented with exceeds the thresholds of 32 kB of RAM and 128 kB of flash memory. Although our prototype could be further optimized, it fits the nature of constrained IoT devices used in the market today.

Extending our measurements to the SUIT manifest case, the code has to be extended with components required by the SUIT specification. A SUIT module and the necessary serialization and cryptographic functions increase the flash size by 10 KB compared to the simple OTA scenario. While the COSE and the CBOR modules are here specifically required for SUIT compliance, in a real-world scenario these modules could also be used for sensor data encoding and application data encryption.

Using LwM2M compatible handlers for this increases the flash size by another 2 kB because of the need to implement the mandatory LwM2M handlers and the registration protocol. These components must be implemented by every device that is LwM2M-compliant and should not be considered as overhead purely related to having over-the-air update functionality.

7.2 Evaluating the Cost of Cryptography

In this section, we evaluate the cost of cryptographic signatures on various constrained IoT devices, with the crypto libraries we surveyed in Section 5. We measure the memory required (flash footprint and stack usage in RAM) and the speed for digital signature verification. We summarized our high-level observations in Table 2 and gave points in the form of ⋆, where fewer points is better. There are tradeoffs between code size, RAM utilization, and speed. For the flash size and the stack size, we take the maximum of the three measured architectures (M0+, M3, and M4). For the verification time, we consider M0+ and M4 only because M3 is somewhere between the two.

Scheme	Library	Cortex M0+	Cortex M3	Cortex M4
ed25519	HACL*	16962	18842	18828
	TweetNaCl	5564	5564	5568
	uNaCl	5572	5528	5536
	C25519	4646	4838	4822
	Monocypher	12632	10334	10358
	WolfSSL	5692	5914	5910
P256r1	TinyCrypt	5048	4888	4900
	Mbed TLS	16660	15356	15378
Other	qDSA	15407	12080	12070
	libhydrogen	2222	2176	2172

Table 6: Flash size for crypto libraries (in bytes) for signature verification.

Table 6 shows flash memory measurements, and Table 7 shows RAM (statically allocated stack) memory measurements. Table 8 shows the speed of signature verification.

Among the libraries that provide ed25519 signature and verification, we observe major differences in terms of performance. C25519 is optimized for a low memory footprint on embedded systems and performs best in terms of flash and stack requirements.While not being specifically optimized for embedded systems, Monocypher performs the fastest ed25519 signature operations on all of the hardware we tested, but requires two to five times more stack and flash memory compared to C25519. HACL*, TweetNaCl, and uNaCl also require consistently more memory than C25519, and are slower than C25519 on Cortex M0+. We note that the HACL* and TweetNaCl libraries are also not yet fully optimized for constrained IoT. On Cortex M4 and Cortex M3, TweetNaCl and HACL* are nevertheless faster than C25519. Looking at overall performance, WolfSSL has an average flash size. The stack requirements are relatively low compared to the other ed25519 implementations, but speed is a bit lacking compared to other libraries. All in all, based on our measurements of ed25519 libraries, C25519 seems like a good compromise.

The P256r1-based ECDSA signature scheme, as implemented by TinyCrypt, outperforms most ed25519 implementations by a large margin in terms of speed and stack usage. On the other hand, the flash requirements are comparable to that of the C25519 library. Mbed TLS requires a bigger flash size largely because of the big number library, which requires 5.7 kB. Mbed TLS outperforms TinyCrypt in terms of speed on the Cortex M0+ platform, but is slower on the other platforms with signature verification. Stack usage is, however, significantly higher than for TinyCrypt but lower than most ed25519 implementations.

Even faster on Cortex M0+, qDSA outperforms all other libraries by an order of magnitude or more. The implementation is optimized with assembly code for use on a Cortex M0+ and for 8-bit AVR but not yet for Cortex M3/M4 where it uses a slower C implementation. This explains why qDSA is slower on Cortex M3/M4. While the qDSA implementation takes a relatively large amount

Scheme	Library	Cortex M0+	Cortex M3	Cortex M4
ed25519	HACL*	3184	3272	3272
	TweetNaCl	3764	3800	3768
	uNaCl	3772	3792	3760
	C25519	976	1048	1012
	Monocypher	5188	5088	5088
	WolfSSL	1296	1328	1328
P256r1	TinyCrypt	604	680	664
	Mbed TLS	792	800	800
Other	qDSA	488	792	972
	libhydrogen	488	472	440

Table 7: Stack size of crypto libraries (in bytes) for signature verification.

Scheme	Library	Cortex M0+	Cortex M3	Cortex M4
ed25519	HACL*	7067	1526	1279
	TweetNaCl	7983	1988	1452
	uNaCl	8086	1804	1495
	C25519	4178	3326	1938
	Monocypher	529	72	45
	WolfSSL	3652	2686	1698
P256r1	TinyCrypt	1149	440	348
	Mbed TLS	1558	1132	838
Other	qDSA	134	1920	1291
	libhydrogen	1061	220	237

Table 8: Signature verification time (in milliseconds).

of flash space, a closer look shows that at least 8 kB are required for a SHA-3 digest algorithm, which is required by qDSA. This cost could be amortized when other parts of the system also use the SHA-3 algorithm.

The Gimli-permutation-based Libhydrogen performs very well in both size and speed on all platforms. While it is not the fastest crypto library, the flash usage and stack requirements are the lowest among the tested libraries.

8 DISCUSSION: GOING FORWARD

A few observations can be made based on our work.

- **State-of-the-art crypto is doable on IoT devices, but it takes a toll**. Widely-used security algorithms are reasonably fast and fit the memory budget on constrained IoT devices. However, crypto consumes a significant chunk of the resources available on such devices. Given an algorithm, memory usage and speed can vary by an order of magnitude, depending on the implementation's tradeoffs. New algorithms, such as qDSA, provide promising alternatives, even faster and smaller. In any case, hardware crypto acceleration should also be considered. The implications of switching to post-quantum crypto algorithms, like hash-based signatures, on constrained IoT devices have to be studied.
- **Making the firmware update reliable is key**. With the system we described, the maintainer is expected to test the new firmware properly before rolling it out. At a minimum, the new firmware must be able to update itself one more time over-the-air. Guarantees beyond this minimum requirement – such as the use of watchdog timers and the ability to use a "factory reset" – fall into the realm of traditional embedded software management and increase the flash memory requirements. Without taking these considerations into account, failures, like those reported with the Taiwanese YouBike service [34] and the Japanese X-ray telescope satellite Hitomi [42], are likely to occur again.
- **Use delegation capabilities with care**. As the system allows the maintainer to transfer its authority to another entity, the maintainer is entrusted with the responsibility of not transferring authority to malicious entities. If the maintainer is the owner of the device, trust is not an issue; otherwise, maintenance of IoT software is typically of a contractual nature, and the caveats of such trust are well-trodden territory. An improvement of the system could use protected memory and/or a dedicated crypto hardware module to validate authority transfer.
- **Shielding against resource exhaustion and best-before vulnerabilities**. The extent to which an IoT device is protected against resource exhaustion attacks depends on the resources of the firmware update server in the LwM2M-OTA configuration. The aspect of dimensioning the server's resources to counter potential DoS attacks is covered by extensive prior work in the domain. In the end, due to extreme lack in resources, constrained IoT devices remain intrinsically vulnerable.
- **Real-world requirements make firmware updates complex**. In this paper we focused our efforts on the most basic scenario outlined in [43] and we did not consider refinements, such as firmware encryption, updating devices with multiple microcontrollers, complications due to policy handling, differential updates, or more efficient distribution using multicast. Encryption, for example, raises the question about key management. In a world where software components are developed, maintained, and updated by different developers, additional challenges arise. While the advantages are known from web development, there are questions about how to trace component versions and their composability with other software libraries, how to sandbox components in constrained IoT devices, how to accomplish faster time to market in regulated industries where software development requirements and testing are much harder than on the internet, and so on.
  We expect a number of these topics to be investigated in the IETF SUIT working group.
- **IoT software updates are not just for critical infrastructure**. Interdependence between networks has dramatically increased over the past few decades. Enabling and securing firmware updates is necessary for IoT devices that are (i) inside the infrastructure perimeter (for example, industrial sensors), and (ii) outside the infrastructure perimeter (for example, consumer smart appliances). For instance, a recent study [59] shows how the power grid is indirectly vulnerable to DDoS attacks from hacked consumer appliances in smart homes. Using simulations, the study shows

how a botnet controlling a relatively small number of connected water heaters and air conditioners could maliciously disrupt power demand and take down most of a large power grid serving an area as large as Canada (tens of millions of people).

- **Firmware update security is more than network security**. Software-based attacks, such as buffer overflow attacks, known from the desktop and mobile world, are also very likely to increase in the IoT world. More work on memory isolation and compartmentalization is required because the most popular IoT operating systems offer only few isolation mechanisms to developers. Hardware-based attacks require attention to be paid to side channel analysis but also to exposed components, such as off-chip flash or debug ports that are left unprotected. Note that the system we described does not protect against tampering of this nature.
  Software supply chain vulnerabilities become important when software bundle components are handled by different developers. Recent attacks have laced legitimate software with backdoors and/or malware, such as the Ccleaner software [28]. In our prototype, the authorized maintainer centralizes the responsibility of assessing the legitimacy of firmware updates. In some cases, it may become difficult for the maintainer to assess this legitimacy, and a decentralized version of the assessment may become necessary [46].
- **Something is better than nothing**. The prototype described in this paper demonstrates how a basic firmware update mechanism with state-of-the-art security can be introduced to constrained IoT devices. This added functionality brings a welcome improvement to the world of unmaintained IoT devices that offer no story for updating buggy software.

9 CONCLUSION

Including a firmware update mechanism in IoT devices is a *must-have* feature. This need is exacerbated by the current context, where cyber-criminality is on the rise, while full-blown, state-driven cyberwars are being fought on a large scale.

In this paper, we have surveyed open standards, which provide generic building blocks for secure firmware updates on constrained IoT devices. We have built a basic prototype, bundling such standard building blocks and avoiding proprietary components as much as possible. We assessed the security characteristics of the resulting system, and we showed how it brings state-of-the-art security to IoT devices. The cost of enabling the firmware update solution in our prototype is bearable, in terms of the required memory and computation, with the currently available IoT hardware. We demonstrate that it is possible to implement a generic, standards-compliant firmware update solution on IoT devices without exceeding the typical thresholds of 32kB of RAM and 128kB of flash memory.

REFERENCES

[1] [n. d.]. SUIT-compliant IoT Firmware Update Prototype. https://github.com/bergzand/RIOT/tree/app/suit-ota/examples/suit_updater
[2] [n. d.]. The MCUboot Bootloader. https://github.com/runtimeco/mcuboot
[3] [n. d.]. The Update Framework. https://github.com/theupdateframework/tuf
[4] [n. d.]. Zephyr Project. https://www.zephyrproject.org
[5] 2016. WIRED Magazine: The Botnet That Broke the Internet Isn't Going Away. https://www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/.
[6] 2017. Curve25519 and Ed25519 for low-memory systems. https://www.dlbeer.co.nz/oss/c25519.html
[7] 2018. Libhydrogen. https://github.com/jedisct1/libhydrogen/
[8] 2018. TinyCrypt Cryptographic Library. https://github.com/01org/tinycrypt
[9] 2018. WolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com/
[10] Francisco Javier Acosta Padilla et al. 2016. The Future of IoT Software Must be Updated. In *IAB Workshop on Internet of Things Software Update (IoTSU)*.
[11] Amazon. [n. d.]. FreeRTOS. https://www.freertos.org/.
[12] Arm. [n. d.]. Mbed OS. https://mbed.org/technology/os/
[13] Arm. [n. d.]. Mbed TLS. https://tls.mbed.org
[14] N Asokan et al. 2018. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).
[15] Emmanuel Baccelli et al. 2018. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE PerCom*.
[16] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. 2018. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal* (2018).
[17] Daniel J Bernstein et al. 2014. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 64–83.
[18] Daniel J Bernstein et al. 2017. Gimli: a cross-platform permutation. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 299–320.
[19] Carsten Bormann et al. 2014. RFC 7228: Terminology for constrained node networks. IETF Request For Comments. http://www.ietf.org/rfc/rfc7228.txt
[20] Carsten Bormann and Paul E. Hoffman. 2013. Concise Binary Object Representation (CBOR). RFC 7049. https://doi.org/10.17487/RFC7049
[21] Broadband Forum. [n. d.]. TR-069, CPE WAN Management Protocol Version 1.4. https://www.broadband-forum.org/technical/download/TR-069.pdf.
[22] Broadband Forum. [n. d.]. User Services Platform. https://usp.technology/.
[23] Stephen Brown and Cormac J Sreenan. 2013. Software updating in wireless sensor networks: A survey and lacunae. *Journal of Sensor and Actuator Networks* 2, 4 (2013), 717–760.
[24] Robert Davis, Nick Merriam, and Nigel Tracey. 2000. How embedded applications using an RTOS can stay within on-chip memory limits. In *12th EuroMicro Conference on Real-Time Systems*. Citeseer, 71–77.
[25] Michael Düll et al. 2015. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography* 77, 2-3 (2015), 493–514.
[26] Adam Dunkels et al. 2006. Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *ACM EWSN*.
[27] Dustin Frisch, Sven Reißmann, and Christian Pape. 2017. An Over the Air Update Mechanism for ESP8266 Microcontrollers. (10 2017).
[28] Andy Greenberg. 2017. Software has a Serious Supply-Chain Security Problem. *Wired* (Sep 2017). https://www.wired.com/story/ccleaner-malware-supply-chain-software-security
[29] HACL*. [n. d.]. Verified C code crypto library. https://github.com/mitls/hacl-c
[30] Oliver Hahm et al. 2016. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal* (2016).
[31] Jonathan W Hui and David Culler. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *ACM Sensys*.
[32] IETF. [n. d.]. Trusted Execution Environment Provisioning (TEE) Working Group. https://datatracker.ietf.org/wg/teep/about.
[33] Jaein Jeong and David Culler. [n. d.]. Incremental network programming for wireless sensors. In *IEEE SECON, 2004*.
[34] Liu Jian-band et al. 2016. YouBike service down in Taiwan. *Focus Taiwan* (Aug 2016). http://focustaiwan.tw/news/asoc/201608310010.aspx http://focustaiwan.tw/news/asoc/201608310010.aspx.
[35] S. Josefsson and I. Liusvaara. 2017. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032. http://www.ietf.org/rfc/rfc8032.txt
[36] Sye Loong Keoh, Sandeep S Kumar, and Hannes Tschofenig. 2014. Securing the internet of things: A standardization perspective. *IEEE Internet of Things Journal* 1, 3 (2014), 265–275.
[37] Trishank Kuppusamy et al. 2018. Uptane: Security and customizability of software updates for vehicles. *IEEE Vehicular Technology Magazine* (2018).
[38] Micrium/Silicon Labs. [n. d.]. µC/OS. https://www.micrium.com.
[39] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, Vol. 37. ACM, 85–95.
[40] Amit Levy et al. 2017. Multiprogramming a 64kb computer safely and efficiently. In *ACM SOSP*.
[41] D McGrew, K Igoe, and Margaret Salter. 2011. *Fundamental elliptic curve cryptography algorithms*. Technical Report.
[42] Rud Merriam. 2016. Software Update Destroys $286 Million Japanese Satellite. https://hackaday.com/2016/05/02/software-update-destroys-286-million-japanese-satellite/. *Hackaday* (May 2016).

[43] Brendan Moran et al. 2019. *A Firmware Update Architecture for Internet of Things Devices*. Internet-Draft. IETF. https://datatracker.ietf.org/doc/html/draft-ietf-suit-architecture-02 Work in Progress.
[44] Brendan Moran et al. 2019. *Firmware Updates for Internet of Things Devices - An Information Model for Manifests*. Internet-Draft. IETF. https://datatracker.ietf.org/doc/html/draft-ietf-suit-information-model-02 Work in Progress.
[45] Brendan Moran and Hannes Tschofenig. 2018. *A CBOR-based Firmware Manifest Serialisation Format*. Internet-Draft draft-moran-suit-manifest-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-moran-suit-manifest-03 Work in Progress.
[46] Kirill Nikitin et al. 2017. CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds. In *USENIX Security*. 1271–1287.
[47] NIST. [n. d.]. Post-Quantum Cryptography Project. https://csrc.nist.gov/projects/post-quantum-cryptography
[48] National Institute of Standards and Technology. 2013. Digital Signature Standard. In *Federal Information Processing Standards FIPS 186-4*. NIST.
[49] OMA. 2018. LwM2M Technical Specification, Approved Version 1.0.2. (Feb 2018). http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/
[50] OMA SpecWorks. 2018. Lightweight Machine to Machine Technical Specification: Core, Approved Version 1.1. (July 2018). http://www.openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/
[51] OMA SpecWorks. 2018. Lightweight Machine to Machine Technical Specification: Transport Bindings, Approved Version 1.1. (July 2018). http://www.openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/
[52] qDSA. [n. d.]. Package on RIOT. https://github.com/RIOT-OS/qDSA
[53] Joost Renes and Benjamin Smith. 2017. qDSA: Small and Secure Digital Signatures with Curve-based Diffie–Hellman Key Pairs. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 273–302.
[54] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. 2017. IoT goes nuclear: Creating a ZigBee chain reaction. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 195–212.
[55] Peter Ruckebusch, Eli De Poorter, Carolina Fortuna, and Ingrid Moerman. 2016. Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. *Ad Hoc Networks* 36 (2016), 127–151.
[56] Jim Schaad. 2017. CBOR Object Signing and Encryption (COSE). RFC 8152. https://doi.org/10.17487/RFC8152
[57] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 7252. https://doi.org/10.17487/RFC7252
[58] Zhengguo Sheng, Shusen Yang, Yifan Yu, Athanasios V Vasilakos, Julie A McCann, and Kin K Leung. 2013. A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities. *Wireless Communications, IEEE* 20, 6 (2013), 91–98.
[59] Saleh Soltan, Prateek Mittal, and H Vincent Poor. 2018. BlackIoT: IoT Botnet of high wattage devices can disrupt the power grid. In *Proc. USENIX Security*, Vol. 18.
[60] Milosh Stolikj et al. 2013. Efficient reprogramming of wireless sensor networks using incremental updates. In *IEEE PERCOM*.
[61] Hannes Tschofenig and Thomas Fossati. 2016. Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925. https://doi.org/10.17487/RFC7925
[62] Loup Vaillant et al. [n. d.]. Monocypher. https://monocypher.org/.
[63] M. Veillette et al. 2018. CoAP Management Interface (CoMI). IETF Internet Draft, https://tools.ietf.org/html/draft-ietf-core-comi-03.
[64] Xiaorui Zhu, Xianping Tao, Tao Gu, and Jian Lu. 2017. ReLog: A systematic approach for supporting efficient reprogramming in wireless sensor networks. *J. Parallel and Distrib. Comput.* 102 (2017), 132–148.
[65] Torsten Zimmermann, Jens Hiller, Jens Helge Reelfs, Pascal Hein, and Klaus Wehrle. 2018. SPLIT: Smart Protocol Loading for the IoT. (2018).
[66] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1789–1806.