2024 IEEE Workshop on Crystal-Free/-Less Radio and System-Based Research for IoT (CrystalFreeIoT)

RobOTAP: Over-the-Air Programming of Robotic Swarms

Alexandre Abadie, Said Alvarado-Marin, Filip Maksimovic, Mališa Vučinić, Thomas Watteyne
Inria, France
e-mail: first.last@{inria.fr}

***Abstract*—Over-the-Air Programming (OTAP) is an essential capability when operating a large low-power wireless deployment such as wireless sensors or swarm robots. OTAP needs to be carefully crafted for these use cases to take into account the limited communication bandwidth, the unreliability and the large latency associated with the low-power wireless nature of the network, as well as the constrained nature of the microcontrollers that are being updated. This paper introduces RobOTAP, an OTAP solution specifically targeted at robotic swarms, available as an open-source implementation. RobOTAP is designed to be minimalistic, with the OTAP module and bootloader having flash footprints below 1 kB / 4 kB, respectively. It is designed to be fast: we show a full update for an 18 kB image in less than 2.3 s. Finally, RobOTAP is secure: we compare on both the nRF52840 and the nRF5340 the performance when using ARM CryptoCell-310 hardware acceleration, a software implementation of the same security routines, or no security at all.**

***Index Terms*—otap, microcontroller, swarm, security, robotics**

I. Introduction

Over-The-Air Programming (OTAP) is a fundamental capability for any embedded firmware deployment. Every embedded system at one point needs to be updated to either fix a bug, patch a security vulnerability or roll out a new feature. Embedded deployments, including wireless networks of low-power sensors, or swarms of robot are composed of hundreds of devices. Being able to reprogram them “over-the-air” is critical for any system operator.

OTAP is an involved mechanism from an embedded programming point of view. It consists of a protocol to transfer a new firmware image into the robot’s micro-controller, coupled with a “bootloader”: a small program which runs a first stage boot and eventually switches to the new image. We focus specifically on OTAP for robotic swarms, which comes with the following challenges: *How can a user update a swarm of microcontroller-based robots without downtime? How can OTAP ensure no data has been altered during the update (integrity)? How can OTAP reject binaries from unknown developers (authenticity)?*

This paper introduces RobOTAP, a versatile and easy-to-use OTAP implementation designed for microcontroller based robotic swarms. RobOTAP has no external dependencies. It is implemented in a “bare-metal” fashion with minimalist memory requirements. The RobOTAP protocol uses a workflow equivalent to that of the IETF SUIT standardized protocol [1], while exhibiting a much smaller footprint and being designed specifically for updating firmware on robot swarms. The RobOTAP bootloader has a footprint below 4 kB, which means it fits in a single flash page on most modern micro-controllers.

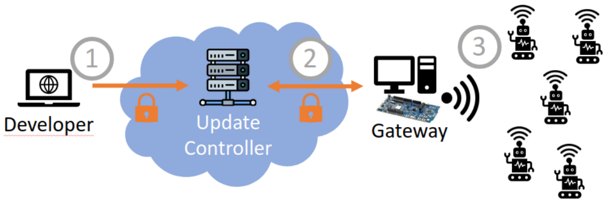


Fig. 1. Infrastructure used for over-the-air programming of a robotic swarm. ① The developer compiles and uploads an update to the OTAP controller. ② The OTAP controller notifies and sends the firmware, chunk-by-chunk, to the swarm gateway. ③ The robots in the swarm apply the update and notify the controller once done.

Fig. 1 illustrates the OTAP process when using RobOTAP. A developer has just finished writing an update version of the firmware for the swarm, and wishes to load that onto all robots. They send that image to the update controller, which is in charge of driving the OTAP protocol. This consists of cutting the image into small enough chunks so they can be sent wirelessly to the robots, The gateway is the networking element that connects the traditional network the update controller and developer are connected to, with the (constrained) wireless network interconnecting the robots. Once the update controller gets confirmation that the new image is loaded onto the robots, it remotely triggers a reset of the microcontroller on the robots. After reset, the bootloader boots the new image.

The contribution of this paper is three-fold:

1) We provide a comprehensive survey of the major OTAP techniques and implementations.
2) We describe RobOTAP, the open-source OATP solution for robotic swarm.
3) We detail the performance one can expect from using RobOTAP, both in terms of memory requirements and duration of the OTAP process.

II. Related Work

There are many ways to update code on microcontrollers. Arakadakis *et al.* [2] propose an exhaustive survey of the common OTAP techniques, discussing security concerns and listing existing software platforms.

979-8-3503-6330-2/24/$31.00 ©2024 IEEE
DOI 10.1109/CrystalFreeIoT62484.2024.00012

A. Software solutions

Multiple open-source operating systems for microcontrollers, such as FreeRTOS [3], RIOT [4], Zephyr [5], provide support for firmware update. FreeRTOS has an official documentation [6] to perform over-the-air (OTA) updates using Message Queue Telemetry Transport (MQTT) [7].

Oliveira *et al.* [8] use Zephyr to evaluate multiple firmware update techniques: traditional full reprogramming, partial programming, scripting and virtual machines.

Baccelli *et al.* [9] describe how to update a JavaScript script running in an embedded interpreter on highly constrained devices (32 kB of RAM) using RIOT. Zandberg *et al.* [10] propose scripting in Femto-container, based on eBPF, and implemented over RIOT.

UpKit [11] is a portable and lightweight library designed to facilitate the implementation of embedded firmware update systems on constrained devices.

Component-based updates, where a sub-part of a binary firmware is updated, are proposed in various frameworks, including Kevoree [12], Elon [13] or Concept-OS [14]. Component-based updates are reducing the bandwidth consumption during the update but are also increasing drastically the complexity level and development constraints for robot programmers.

B. Bootloaders

The bootloader is the core piece of a full firmware image update process for which many open-source implementations exist.

RIOT implements its own bootloader, called `riotboot` which consumes only 4 kB of flash memory. This bootloader requires image metadata to be prepended at the very beginning the actual firmware image, making it hard to use arbitrary toolchains for programming IoT devices.

MCUBoot [15] is a widely used, full featured and open-source bootloader for versatile and secure boot. The flash requirements of MCUBoot are high, over 32 kB on Flash and, unlike `riotboot`, the image metadata, called *image trailer*, is placed at the end of the firmware image.

WolfBoot [16] is another multi-purpose bootloader with a clear focus on security. WolfBoot is released under the GPL-2 license and also requires 32 kB on flash.

MCUBoot and WolfBoot large flash requirements are explained by their long list of features, most of which are not of interest for our autonomous robots, like a USB stack. RIOT `riotboot` is the smallest bootloader but suffers the image metadata placement limitation.

C. IETF SUIT

The IETF has approved a general architecture for secure over-the-air updates for IoT devices in RFC9019 called "SUIT" (Software Updates for Internet of Things) [1]. Zandberg *et al.* [17] propose an implementation of SUIT based on RIOT [4] and compare in detail many cryptographic software libraries. Sahlmann *et al.* [18] propose MUP an implementation of SUIT based on the UpKit library and using the MQTT [7] protocol. Unfortunately, hardware based cryptographic accelerators are not taken into account by both work.

In this paper we introduce RobOTAP, which distinguishes itself from the solutions listed above *(1)* by the fact that it is built specifically for updating swarms of robots and *(2)* by its minimalist nature, allowing it to run on any platform with close to no overhead.

III. RobOTAP

A. Overview

Fig. 1 illustrates the general infrastructure of RobOTAP. There are four main parts:

- The **developer** or **swarm maintainer** writes the software for the robots in the swarm.
- The **OTAP controller** is a trusted server running in the cloud that is interacting with the swarm to perform the firmware update. The OTAP controller tracks the status of the deployed firmware images in the swarm.
- The **swarm gateway** is a computer that interconnects the communication medium of the robots, typically wireless, to the Internet. The gateway is transparent as it is just a routing element of the network.
- The **swarm robots** receive firmware update messages from the OTAP controller and acknowledge them. The robots periodically advertise the OTAP controller about status information containing their current firmware version and other firmware setup.

In this paper, we consider the communication protocol that connects the developer to the OTAP server, and the OTAP server to the swarm gateway, as secure, e.g. authenticated and encrypted.

Between these four entities, a typical firmware update operates as follows:

1) When a new firmware version is ready to be deployed, the developer sends it to the OTAP controller.
2) The OTAP controller broadcasts the update to the robots in the swarm, via the gateway.
3) The OTAP controller notifies the developer once the firmware update has completed.

B. Firmware Update Mechanism on a Microcontroller

A bare-metal application running on a microcontroller is self-contained, in the sense that it's not running within the context of another software project. It embeds all the code required for initializing and managing the hardware, along with its specific application logic. Also, in contrast to applications running on regular high-level computers that can use the hardware concurrently, only one standalone application can run at a time on a microcontroller. As a result, to update a complete firmware, it is necessary to split the non volatile memory into several partitions, where each partitions contains a single application. And to choose which application to boot, a bootloader partition is required at the very beginning of the flash memory.

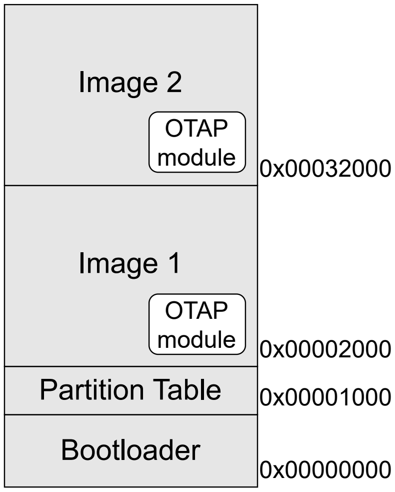


Fig. 2. Memory Layout used for the firmware update.

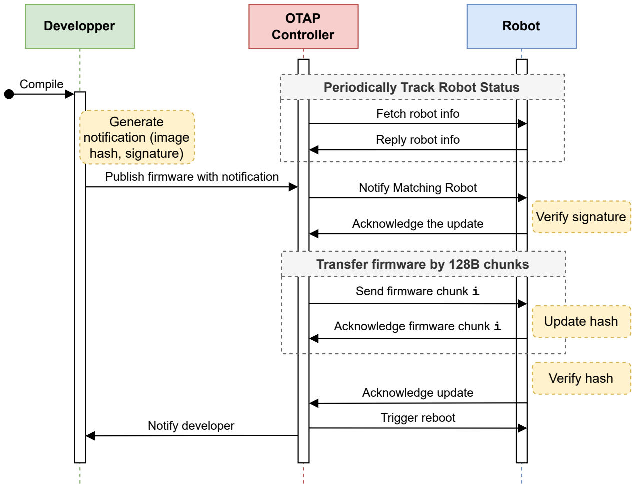
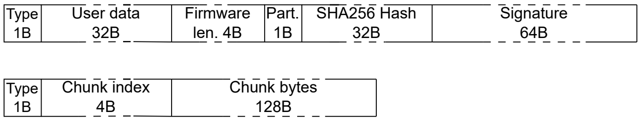


Fig. 3. Sequence diagram of the RobOTAP protocol.



Type 1B	User data 32B	Firmware len. 4B	Part. 1B	SHA256 Hash 32B	Signature 64B

Type 1B	Chunk index 4B	Chunk bytes 128B

Fig. 4. Update notification (top) and single firmware chunk (bottom) packets

Fig. 2 shows a possible memory layout that can be used with firmware updates. The bootloader resides at the beginning of the flash and is the entry point when the microcontroller boots. The remainder of the flash memory is divided into two partitions of the same size, and contains two versions of the main firmware image. One flash page between the bootloader and the first image is used to store the partition table that describes the memory layout. The partition image is read by the bootloader to learn which image has to be booted and whether it can be booted at all. Although this design wastes some flash space, i.e. a complete flash page, so 4 kB on the popular nRF52/nRF53 series of microcontrollers, it allows this bootloader to be used with firmware built from any software solutions or operating systems.

When provisioning the device, the flash memory must be programmed with the bootloader and a valid image on the first partition. Each image contains the logic, that we call “OTAP module” in Fig. 2, to receive a firmware update over-the-air.

A typical firmware update workflow on the microcontroller is as follows:

1) The bootloader reads the partition table to check what is the active partition, and boots this image.
2) When the firmware update starts, the active image writes the received chunks in the other partition.
3) Once completed, the active image updates the active index in the partition table and reboots.
4) Upon reboot, the bootloader reads the partition table and boots the newly downloaded image on the second partition.

To properly work with such memory layout, the generated application image must have position dependent binary code. Thus, special care must be taken by the developer to ensure the build system’s linker scripts use compatible start address and flash sizes. For example, in Fig. 2, partition 1 starts at address `0x0002000`, partition 2 at address `0x0032000`. A firmware linked for partition 1 and flashed on partition 2 cannot run. If the flash size is greater than the target partition table, it might allow to link although the resulting firmware does not fit in memory.

C. Protocol

Microcontrollers have limited internal memory, in the order of hundreds of kB, and use constrained communication media. For example, a 2.4 GHz BLE radio can carry at most 255 bytes long payloads. Therefore, a complete firmware image cannot be sent in one go to a microcontroller and must be split in small chunks. As a result, a protocol has to be defined in order to drive the over-the-air firmware update between the OTAP controller and the robots. This protocol serves several purposes:

- Identify the **version** and the **current active partition** used by the application running on the robot. Knowing the active partition index is critical to prevent a firmware linked for one partition to be written on the other one. Knowing the version is also important to avoid triggering an unnecessary update.
- Verify the **authenticity** and **integrity** of the downloaded firmware on the microcontroller side.
- Wrap the firmware update transfer with a **signaling layer**, used to notify the start of the update process, used by each robot to acknowledge firmware chunks received, and used by the controller to notify robots when they can reboot on the image.

As illustratred in Fig. 3, after compiling a new firmware version, the developer starts by creating an update notification message and publishes it, along with the new firmware image, to the OTAP controller. This notification message, shown in

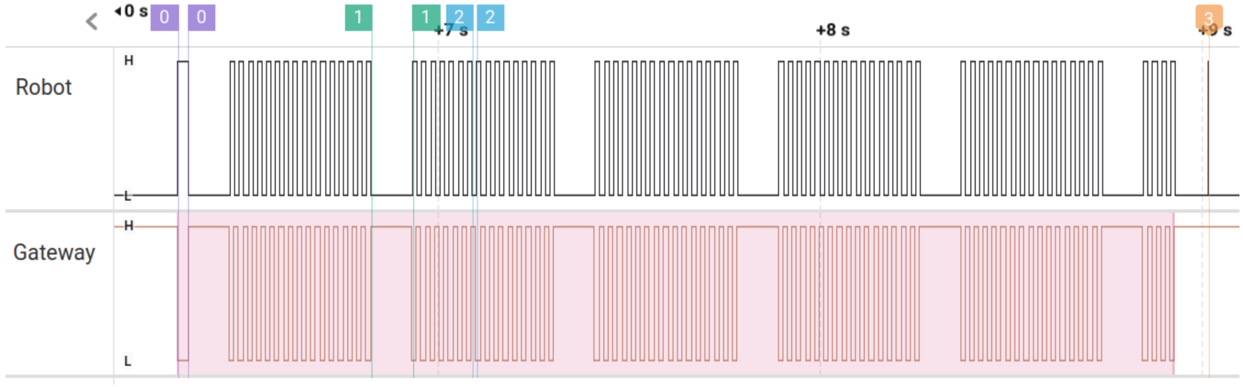


Fig. 5. Measurements from a logic analyzer connected to the swarm gateway and the robot running an nRF52840 with CryptoCell hardware accelerator enabled. Markers 0 (purple) shows the signature verification and the reception and writing of the first chunk (28 ms). Markers 1 (green) show the flash page erase, and writing of one chunk (108 ms). Markers 2 (blue) show the reception, and writing of one chunk (11 ms). Markers 3 (yellow) show the moment the robot reboots. We highlight red the duration of the full firmware update process.

Fig. 4 contains firmware metadata: firmware info (custom user defined string), firmware size, target partition, firmware hash, and metadata cryptographic signature. The necessary cryptographic primitives for implementing our protocol are therefore a hash algorithm and a digital signature algorithm.

To start the update, the OTAP controller initially sends the update notification message to robots matching the firmware info, target partition and firmware version. Upon receiving that notification, the robots verifies the signature; if it is valid, they send an acknowledgement back to the OTAP controller.

The OTAP controller then starts sending 128 byte chunks (see Fig. 4) of the binary firmware. For each chunk received, the OTAP module writes the chunk directly to the flash after the previously written chunk, updates the received firmware hash, and replies back an acknowledgement message to the OTAP controller.

When the last chunk is received by the robot application, it finalizes the hash of the bytes received and compares it with the hash initially received in the (signed) update notification message. If they match, this last chunk is acknowledged and the OTAP controller sends a reboot command to the robot update module which in turn, updates the partition table and triggers a reboot on the new firmware version.

Fig. 5 shows a logic analyzer output during a firmware update on an nRF52840-DK board. The OTAP module is configured to use hardware-accelerated cryptography with the ARM CryptoCell peripheral. The output shows that the signature verification with ARM CryptoCell is quite fast, completed in a few ms. The rising/falling edges correspond to firmware chunks being received and written to flash memory. The periodic long delays corresponds to flash page erase operations, when a chunk has to be written at the beginning of the new page. The last peak is triggered when the firmware reboots and initializes the debug pin used with the logic analyzer.

In our use case, the protocol between the gateway and the robots operates directly on top of the 2.4 GHz GFSK physical layer used for BLE, i.e. without a MAC layer, and no multi-hop mesh routing is used. Nevertheless, the RobOTAP protocol is independent from the transport protocol, so it can be used on top of any more complex networking stacks and/or radio protocols.

D. *OTAP Security Considerations*

The firmware image is large compared to the available RAM memory of the microcontroller. Saving the firmware image in RAM while chunks are received is therefore not possible. The remaining option used in the current design is to write chunks into the flash memory of the inactive partition as they are received. The hash of the firmware image can only be computed and verified once all the chunks have been received. An attacker that is able to inject or alter firmware chunks can perform a resource exhaustion (e.g. energy) attack on the robot by making it write invalid chunks into flash memory. This will be detected only once the final chunk has been received and the hash verification fails. A potential mitigation for this attack would be to include several intermediate hashes in the (signed) notification message. This would allow the attack to be detected before the final chunk and discard the remaining chunks.

An attacker with access to the networking packets exchanged between the OTAP Controller and the robots is able to replay the notification message at a later time and trigger the firmware update to a potentially old version of the firmware. Such attacker can inject old legitimate firmware chunks and make the robots believe they are fresh. This downgrade attack works because, in the current design, there are no freshness checks or protection of the messages exchanged between the OTAP Controller and the robots. Including timestamps in the (signed) notification message is a simple mitigation, but requires the robots to have synchronized time with the OTAP Controller. As an alternative, the notification message may contain a monotonically increasing counter, which each robot tracks and rejects previously received messages.

TABLE I
MEMORY FOOTPRINT & OTAP DURATION

Chip	Security	flash / RAM footprint				OTAP duration		
		OTAP module	security	**RobOTAP footprint**	image size*	signature	transfer	**total**
nRF52840	*None*	566 B / 348 B	0 B / 0 B	**566 B / 348 B**	5902 B	0 s	0.851 s	**0.852 s**
	software library	704 B / 348 B	5806 B / 112 B	**6510 B / 460 B**	11906 B	2.798 s	1.423 s	**4.306 s**
	hardware (CrytoCell-310)	704 B / 348 B	11557 B / 256 B	**12261 B / 604 B**	17657 B	0.018 s	2.170 s	**2.273 s**
nRF5340	*None*	564 B / 348 B	0 B / 0 B	**564 B / 348 B**	9627 B	0 s	1.214 s	**1.301 s**
	software library	706 B / 348 B	5814 B / 112 B	**6520 B / 460 B**	15667 B	2.164 s	1.893 s	**4.145 s**

* includes the entire application well beyond RobOTAP.

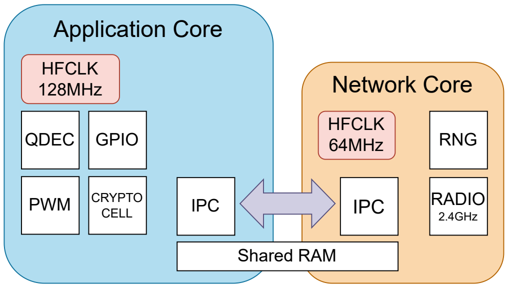


Fig. 6. Nordic Semiconductor's nRF5340 is an asymmetric ARM dual Cortex-M33 cores.

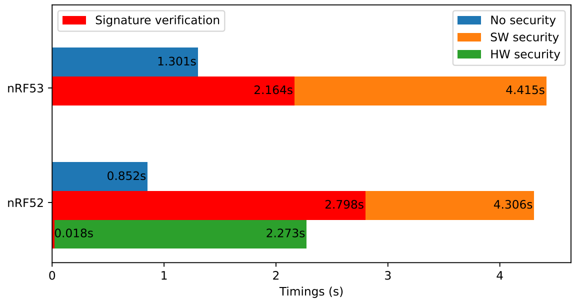


Fig. 7. Firmware update durations using different cryptographic strategies: no cryptography, software cryptography and hardware cryptography based on ARM CryptoCell (only for the nRF52840 for the moment). The signature duration is displayed in red over the total durations.

IV. IMPLEMENTATION

An implementation of RobOTAP is published under a BSD license and is part of the DotBot project[1]. The `DotBot-firmware` repository provides the bootloader code and sample applications for partition 0 and partition 1. RobOTAP is documented in [19].

We use the hash algorithm SHA256 [20] and the signature algorithm Edwards-curve Digital Signature Algorithm (EdDSA), over Ed25519 curve with SHA-512 [21], to instantiate the cryptographic primitives used in the design. The developer has to generate a key pair once, where the private key is used to sign the notification message and must remain secret, while the public key, used by the OTAP module to verify the signature, is built in the firmware.

On an nRF52840 microcontroller, the complete OTAP module has a footprint of 566 B of flash without cryptography, 6510 B of flash with software cryptography and 12,261 B with ARM CryptoCell. When built with internal flash over serial and LEDs user feedback, the bootloader fits in less than 4 kB of flash. In its minimal configuration, e.g. with only partition table parsing and booting code, the bootloader takes less than 2 kB on flash.

V. PERFORMANCE EVALUATION

To evaluate the performance of the OTAP implementation, we measure different timings during a firmware update process between an OTAP controller and a single robot. The measurements are performed using two different target types:

- an nRF52840-based board, with an ARM Cortex-M4 CPU running at 64 MHz, with 256 kB of RAM memory and 1 MB of flash memory. The nRF52840 also embeds an ARM CryptoCell-310 accelerator.
- an nRF5340 based board, which embeds a dual core ARM Cortex-M33. The main core, the application core, runs at 128 MHz, with 256 kB of RAM and 1 MB of flash, and embeds an ARM CryptoCell-312 hardware accelerator. The network core runs at 64 MHz and interfaces to the radio. By default, the nRF5340 is configured to run within the ARM TrustZone security layer, where the application core is part of the secure domain and the network of the non secure domain. As shown in Fig. 6, both cores can exchange data via a shared RAM space and dedicated inter-process communication (IPC) peripherals.

To transfer data over the radio, the nRF microcontroller radio uses 2.4 GHz BLE 1 Mbps mode. The firmware is built in release mode, e.g. without debug symbols, using GCC level 2 optimization. We evaluate the following scenarios:

1) **No cryptography** involved during the firmware update process: the OTAP controller just sends clear metadata and no signature or hash verification is performed on the microcontroller.
2) **Software cryptography** is used on the microcontroller. For the SHA256 hashing function, we use a public domain implementation [22]. For the Ed25519 signature, we use the implementation from [23].
3) **Hardware cryptography** using the ARM CryptoCell cryptographic accelerator. For now, we only have sup-

[1] DotBot GitHub repository: https://github.com/dotbots/DotBot-firmware.

port for the ARM CryptoCell-310 so this scenario is evaluated on the nRF52840 only.

The user code is the same for all scenario, so the only code that differs and that is changing the generated binary size are the cryptographic strategies described above and the board support code.

The OTAP duration is measured from when the OTAP controller sends the update notification, until the device verifies the firmware hash.

Duration results are displayed in Fig. 7. Table I lists fine-grained timings of the main update steps and footprints of the firmware images.

As expected, the software-based signature verification takes the most time, 65% of the entire OTAP duration on the nRF52840, 52% on the nRF5340. The fact that the application core of the nRF5340 runs twice as fast as the nRF52840 microcontroller reduces the signature verification duration by 23%. Please note that the CPU is blocked during the signature verification routine and cannot undertake other tasks.

Using the ARM CryptoCell accelerator reduces that time drastically, to less than 20 ms. This performance gain comes with a 48% footprint increase when compared with the software cryptography. This is because the ARM CryptoCell driver is an ARM proprietary blob that contains unnecessary code such as more hashing algorithms, asymmetric/symmetric encryption, or elliptic curves cryptography, and it is not possible to choose only what is needed to reduce that size. Nevertheless this feature is essential to perform a secure firmware update of a robotic swarm with minimal downtime and latency.

VI. Conclusion

This paper introduces RobOTAP, an OTAP solution designed for robotic swarms. We describe the components and general workflow required to perform a complete firmware update on a microcontroller, explain the RobOTAP protocol in detail, and discuss some security concerns relative to this protocol. We also give performance measurements in different scenarios of our implementation, relative to generated binary code sizes and to update timings. These measurements show a trade off between security requirements and the amount of bytes transferred over-the-air. For very low-latency and maximum responsiveness, using the ARM CryptoCell cryptographic accelerator looks very appealing.

We are working on adding support for the ARM CryptoCell-312 on the nRF5340. We are also building a large scale testbed dedicated to research in robotic swarms, in which RobOTAP plays a key role as users are able to OTAP their firmware remotely. We plan to extend RobOTAP with support for the ARM Cortex-M33 TrustZone environment, which is part of the nRF5340. This provides security partitioning at runtime and allow any arbitrary code from our testbed users to be flashed other-the-air on the robots. In addition, RobOTAP will provide low-level mechanisms, such as bootloader watchdog and trusted backup firmware, to recover the robots from any broken state.

VII. Acknowledgment

This document is issued within the frame and for the purpose of the OpenSwarm project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046. Views and opinions expressed are however those of the author(s) only and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] B. Moran, H. Tschofenig, D. Brown, and M. Meriac, "A Firmware Update Architecture for Internet of Things," RFC 9019, 2021.

[2] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware Over-the-air Programming Techniques for IoT Networks – A Survey," *ACM Compututing Surveys*, 2021.

[3] https://www.freertos.org.

[4] https://riot-os.org.

[5] https://www.zephyrproject.org/.

[6] https://www.freertos.org/freertos-core/over-the-air-updates/index.html.

[7] C. Sengul and A. Kirby, "Message Queuing Telemetry Transport (MQTT) and Transport Layer Security (TLS) Profile of Authentication and Authorization for Constrained Environments (ACE) Framework," RFC 9431, 2023.

[8] J. Oliveira and F. Sousa, "Reprogramming of Embedded Devices using Zephyr: Review and Benchmarking," in *Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, 2021.

[9] E. Baccelli, J. Doerr, S. Kikuchi, F. A. Padilla, K. Schleiser, and I. Thomas, "Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2018.

[10] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers," in *ACM/IFIP International Middleware Conference*, 2022.

[11] A. Langiu, C. A. Boano, M. Schuß, and K. Römer, "UpKit: An Open-Source, Portable, and Lightweight Update Framework for Constrained IoT Devices," in *International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[12] F. J. Acosta Padilla, F. Weis, and J. Bourcier, "Towards a Model@runtime Middleware for Cyber Physical Systems," in *Workshop on Middleware for Next Generation Internet Computing (MW4NG)*, 2014.

[13] W. Dong, Y. Liu, C. Chen, L. Gu, and X. Wu, "Elon: Enabling Efficient and Long-term Reprogramming for Wireless Sensor Networks," *ACM Transactions on Embedded Computing Systems*, 2014.

[14] A. Aspesi and V. Zaccaria, "ConceptOS: A Micro-Kernel Approach to Firmware Updates of Always-On Resource-Constrained Hubris-Based IoT Systems," *IEEE Internet of Things Journal*, 2023.

[15] https://docs.mcuboot.com/.

[16] https://github.com/wolfSSL/wolfBoot.

[17] Zandberg, Koen and Schleiser, Kaspar and Acosta, Francisco and Tschofenig, Hannes and Baccelli, Emmanuel, "Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check," *IEEE Access*, 2019.

[18] K. Sahlmann, V. Clemens, M. Nowak, and B. Schnor, "MUP: Simplifying Secure Over-the-air Update with MQTT for Constrained IoT Devices," *MDPI Sensors*, 2020.

[19] https://dotbot-firmware.readthedocs.io/en/latest/otap.html.

[20] T. Hansen and D. E. r. Eastlake, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234, 2011.

[21] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, 2017.

[22] https://github.com/B-Con/crypto-algorithms.

[23] https://github.com/ncme/c25519.