Incremental firmware update over-the-air for low-power IoT devices over LoRaWAN

Internet of Things 34 (2025) 101772

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Research article

Incremental firmware update over-the-air for low-power IoT devices over LoRaWAN

Andrea De Simone, Giovanna Turvani, Fabrizio Riente *

Politecnico di Torino, Department of Electronics and Telecommunications, Italy

ARTICLE INFO

Keywords:
Internet of Things
LoRaWAN
FUOTA
Incremental update
Low-power

ABSTRACT

Remote firmware updates in Internet of Things (IoT) devices remain a major challenge due to the constraints of many IoT communication protocols. In particular, transmitting full firmware images over low-bandwidth links such as Long Range Wide Area Network (LoRaWAN) is often impractical. Existing techniques, such as firmware partitioning, can alleviate the problem but are often insufficient, especially for battery-powered devices where time and energy are critical constraints. Consequently, physical maintenance is still frequently required, which is costly and impractical in large-scale deployments. In this work, we introduce *bpatch*, a lightweight method for generating highly compact delta patches that enable on-device firmware reconstruction. The algorithm is explicitly designed for low-power devices, minimizing memory requirements and computational overhead during the update process. We evaluate *bpatch* on 173 firmware images across three architectures. Results show that it reduces update payloads by up to 39,000×for near-identical updates and by 9–18×for typical minor revisions, eliminating the need to transmit full firmware images. Experimental results further demonstrate significant time and energy savings, with performance comparable to more complex alternatives. *bpatch* is released as open-source and, although demonstrated on LoRaWAN, the approach is flexible and can be adapted to other IoT communication technologies.

1. Introduction

In the IoT scenario, system reliability and durability are essential to ensure efficient and long-term operation. Maintenance activities are often necessary throughout the lifecycle of IoT devices, making device maintainability a key consideration already at the design stage. A defining characteristic of IoT devices is their autonomy and self-sufficiency. Since they are frequently deployed in remote or difficult-to-access locations, or in large-scale deployments, manual maintenance of individual devices is often impractical. Firmware updating represents one of the most common and critical maintenance tasks. Updates may be required for several reasons, including bug fixes, performance enhancements, or adaptation to new operational requirements. Among the various communication technologies, the LoRaWAN protocol is widely recognized as a standard for low-power, long-range data transmission [1]. The Things Network, one of the most adopted LoRaWAN infrastructures, provides support for remote firmware updates through the Firmware Update Over-the-Air (FUOTA) package [2]. LoRaWAN's main strengths lie in its energy efficiency and extensive communication range, up to tens of kilometers [3]. However, its limited bandwidth poses a significant challenge when transmitting large payloads, such as complete firmware images. The traditional FUOTA method addresses this limitation by fragmenting the firmware into smaller packets and transmitting them sequentially. This procedure, however, is both time- and energy-intensive. Numerous studies have

* Corresponding author.
E-mail address: fabrizio.riente@polito.it (F. Riente).

https://doi.org/10.1016/j.iot.2025.101772
Received 8 July 2025; Received in revised form 14 September 2025; Accepted 22 September 2025
Available online 27 September 2025

attempted to improve the remote firmware update process [4]. Analysis of power consumption in LoRaWAN-enabled microcontroller units (MCUs) has shown that the radio transceiver is the dominant energy consumer during FUOTA [5,6]. Class C devices are often used due to the high data throughput required, even though they consume more energy than the more efficient Class A devices, as their radios remain constantly active to receive packets [7]. Another relevant factor is the Spreading Factor (SF), which directly impacts the power and time required for data reception [8]. A practical way to reduce energy consumption is to transmit only the delta between the old and new firmware. Not only must this delta script be compressed before transmission to minimize airtime, but the reconstruction algorithm must also be deliberately simple, so that applying the patch fits within the MCU's limited CPU and memory resources. Developing an algorithm that balances strong delta compression with lightweight reconstruction is therefore non-trivial.

This paper, proposes a novel approach to perform incremental updates, named *bpatch*, which is open-accessible at [9]. The goal is to design a hardware-independent and lightweight solution suitable for integration into low-resource IoT devices. Unlike other techniques that rely on compression algorithms or firmware structure constraints, *bpatch* only uses two simple fundamental codes to *COPY* bytes from the old version or *ADD* bytes from the new one, encoded at the bit-level for minimal overhead. This simplicity enables efficient delta generation and firmware reconstruction without sacrificing compression performance. Importantly, this work focuses on the application of the delta patch, not on the generation process or the security/authenticity aspects of the update procedure. Issues such as secure patch delivery, cryptographic verification, or delta algorithm design fall outside the scope of this paper and are not addressed here.

To validate the effectiveness of *bpatch*, we benchmarked it against popular open-source delta algorithms, including *bsdiff* [10], *VCDIFF* [11] and *HDiffPatch* [12]. These algorithms are generally employed to compress Android Package Kit (APK) files for mobile applications and are unsuitable for hardware-constrained MCUs. Results indicate that *bpatch* achieves compression sizes comparable to these more complex algorithms. Consequently, this method enables effective FUOTA on battery-operated IoT devices, with limited resources, by significantly reducing energy usage associated with firmware transmission and the complexity of reconstruction. Furthermore, we provide an actual implementation of *bpatch* and evaluate its energy savings compared to traditional FUOTA methods in a real-world scenario. The primary contributions of the manuscript include:

- the design of an innovative methodology for efficiently encoding an edit script without relying on additional compression algorithms;
- the creation of a hardware-independent reconstruction algorithm that requires minimal MCU hardware resources;
- the proposed algorithm was benchmarked using 173 delta scripts, derived from real firmware projects, across three different architectures to assess its compression capability and hardware-independence;
- the experimental deployment of *bpatch* in an ultra-low-power IoT node, which includes the STM32WL55JC MCU, demonstrating reduced update duration and energy usage compared to standard updates;
- the public release of the *bpatch* algorithm on Zenodo [9] with a ready-to-use application to generate patches and reconstruct the final firmware.

The rest of this article is organized as follows. A review of energy-aware strategies to update IoT systems is detailed in Section 2. Section 3 overviews an authentic architecture where *bpatch* is implemented. Section 4 presents a detailed description of the developed algorithm. In Section 5, we evaluate the performance of *bpatch* against other open-source solutions. Section 6 examines the energy consumption of FUOTA with and without *bpatch*, and finally, Section 7 provides the conclusions.

2. Related works

Implementing Over-the-Air (OTA) updates in low-power IoT devices presents significant challenges. Several approaches have been proposed to address these issues, with particular emphasis on reducing energy consumption. A comprehensive survey of software updating techniques for wireless sensor networks is presented in [13], which emphasizes the importance of enabling remote management and reviews several dissemination and traffic reduction strategies aimed at decreasing network load. A specific implementation of OTA mechanisms in IoT is reported in [14]. It exploits the Internet Engineering Task Force (IETF) architecture, while [15] proposes an update framework for wireless sensor networks powered by batteries, where energy constraints must be carefully considered. A complementary perspective is provided by [16], which discusses the general challenges of OTA updates and outlines potential approaches for IoT devices. Furthermore, [17] shows that not all operating systems for embedded systems natively support OTA updates, highlighting that this remains an open field with considerable room for improvement. LoRaWAN has emerged as a protocol particularly suited for IoT applications, where energy efficiency and scalability are crucial requirements, as discussed in [18]. Its great scalability is further evidenced in [19], although this comes at the cost of reduced communication reliability. Further analysis in this direction is provided in [20], which illustrates the additional network load introduced by confirmed messages. One of the advantages of LoRaWAN is its support for multicast communication, which enables the broadcasting of messages to multiple nodes and allows firmware dissemination to reduce network congestion during OTA updates [13,21]. Multicast optimization is investigated in [22], where grouping devices by SF improves both energy efficiency and reliability. In [23], unicast and multicast solutions are compared, highlighting the advantages of the latter in terms of update time. Other works in the same direction include [24,25]. Security aspects of firmware updates have also been considered. A blockchain-based solution is proposed in [26] to strengthen version control, though at the cost of higher service overhead and additional registration requirements for nodes and firmware images. In contrast, [24] introduces a cryptographic solution based on Advanced Encryption Standard Cipher Message

Table 1
Overview of main OTA firmware update approaches based on traffic reduction.

Ref.	Optimization	Pro	Cons	Test
[25]	delta encoding (suffix array)	hardware-independent, fast reconstruction	low compression, high memory requirements	10
[29]	No. of flash operations	update reliability	hardware-dependent, low compression	5
[30]	dynamic linking	high compression, no reboot	non-standard compiler, high RAM usage	5
[31]	delta encoding, static allocation	high compression, fast reconstruction	poor flexibility, low scalability	5
[32]	delta encoding	good compression	hardware-dependent	9
[33]	delta encoding (XOR)	hardware-independent, fast reconstruction	low compression, low scalability	7
[34]	delta encoding (diff)	good compression	hardware-dependent	4
[35]	delta encoding (RMTD)	good compression, hardware-independent	complex algorithm	10

Authentication Code (AES-CMAC), which is an extension of AES-128 already integrated into LoRaWAN, and has the advantage of not impacting overall energy consumption. Meanwhile, [27] proposes an additional lightweight cryptographic mechanism based on dual-XOR to further enhance the basic AES-128 scheme used in LoRaWAN.

Several studies have pointed out that transmitting an entire firmware image over LoRaWAN is critical due to bandwidth constraints. As shown in [13,21], the most effective way to mitigate this issue is to transmit only the binary differences rather than the entire image. The study in [25] uses a differential algorithm based on a Suffix Array to generate a delta script for reconstructing the new firmware from the old one. An improvement of this implementation in presented in [28] which incorporates an effective securing mechanism. It employs two standard commands, *COPY* and *ADD* (referred to as *KEEP* and *UPDATE*), and a third one called *LABEL*, which collects the most frequently used opcodes to avoid repetition in the script. The authors claim an energy efficiency improvement of up to 2.65× . The work in [29] presents a solution tailored for energy-harvesting devices, the update system focuses on optimizing the number of flash operations during code replacement. In this case, bandwidth is less critical because the Bluetooth protocol is used, while the main challenge lies in making the system resilient to power interruptions. On the contrary, dynamic code linking is explored in [30]. In this case, the firmware is not monolithic but includes a microsystem capable of loading modules at run-time. Updating one module at a time without requiring a reboot is possible. The main disadvantages are that a standard compiler cannot be used and that dynamic linking generally requires more RAM. The method proposed in [31] compresses the patch size by allocating the functions in static addresses to reduce the effect of address shift. In this case, minor updates result in minimal changes in the new firmware image. In [32], the reconstruction is based on four opcodes: *INSERT*, *MODIFY*, *DELETE*, and *COPY*. These commands are optimized for an 8-bit MCU, where the instruction set is simpler. The work in [33] proposes a differential algorithm based on XOR to increase the speed of the delta script generation process. The commands used are simply *ADD* and *COPY*, making reconstruction faster. The use of *UNIX diff* and variable-length opcodes is analyzed in [34]. The edit script is produced with a variant of this application. To further improve compression, two strategies are adopted: the use of a field to express the length in bytes of the *ADD* and *COPY* opcodes, and the introduction of two additional commands, *REPAIR*, which copies a segment with minor modifications, and *PATCH LIST*, which addresses the problem of address shifts. These additional commands allow a compression rate of up to 92% when code shifting is significant. Finally, in [35], a variant of the Rsync algorithm based on the Reprogramming with Minimal Transferred Data (RMTD) is used to produce a compact patch, achieving an average patch size that is 60% smaller than that of standard Rsync.

Table 1 reports the summary of the main approaches adopted in the literature which aims to reduce the amount of data to transmit. The table indicates the main "Optimization" feature, along with their advantages and disadvantages, "Pro" and "Cons". Last column "Test" indicates the number of different binary pairs where they are employed. The optimizations often rely on assumptions specific to a given MCU and are therefore labeled as *hardware-dependent*. In other cases, even when no hardware assumptions are made, the benchmarks are limited to a small number of firmware images, without comparisons across different architectures to demonstrate true hardware independence. Overall, Table 1 clearly highlights delta encoding as the most effective strategy for achieving a solution that is not tied to a specific architecture. Nevertheless, the choice of binary differencing algorithm plays a critical role, as achieving hardware independence, high compression, and fast reconstruction at the same time remains a significant challenge. Our proposed approach is specifically designed to address this challenge.

3. System architecture and FUOTA integration with *bpatch*

Firmware updates in low-power IoT devices introduce a critical trade-off between functionality and energy efficiency. Traditional FUOTA mechanisms transmit entire firmware images, requiring prolonged radio activity that directly contradicts the ultra-low-power

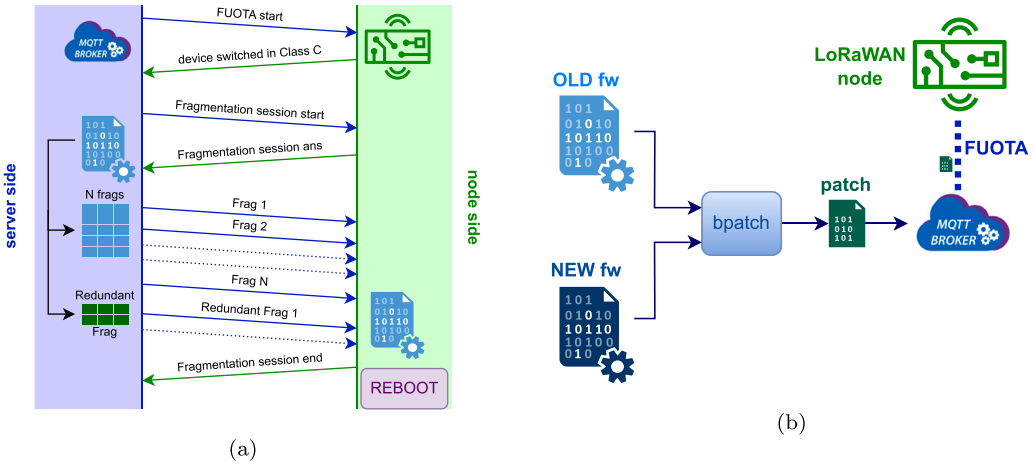

Fig. 1. Overview of the FUOTA protocol (a) and illustration of *bpatch* integration in FUOTA (b).

operational model of typical IoT applications. This section presents the real-world implementation of a FUOTA-enabled architecture where the proposed *bpatch* algorithm has been integrated to address these challenges. To provide a concrete context, we consider an IoT system for monitoring the health status of bee hives [36]. In this application, firmware updates are rare but unavoidable, and energy preservation is paramount. The MCU belongs to the STM32WL family, designed for ultra-low power operation and native LoRaWAN integration. The LoRaWAN protocol was chosen because of the strict energy requirements of the system, as it enables minimal energy consumption at the cost of limited bandwidth. Based on the guideline provided by ST Microelectronics [37], we successfully deployed a standard FUOTA process for supporting complete firmware replacement. The LoRaWAN node operates at 868 MHz with SF 9, providing a maximum payload size of 115 bytes per message. Due to Fair Use Policy limitations on duty cycles, a maximum of 530 downlink messages per hour is allowed, approximately one message every 7 s. The transmission of packets does not require acknowledgment. Lost fragments are recovered using redundant fragments transmitted on demand: after sending all initial fragments, additional redundancy fragments continue to be transmitted until the node confirms successful firmware assembly. The server-side management is performed via a Message Queuing Telemetry Transport (MQTT) Broker, responsible for sending commands, including fragment transmissions, and receiving node responses. A summary of the communication protocol is illustrated in Fig. 1(a), the server side is represented in blue, and the blue arrows indicate the downlinks sent to the node. In contrast, the node is represented in green, and the green arrows indicate the uplinks. Initially, the broker requests the start of the FUOTA session, after which the node switches to Class C. At this point, the broker transmits all firmware fragments, including redundant ones if needed. Once all fragments have been received, the node reconstructs the complete firmware code, and a Firmware Update Agent (FUA) initiates a reboot using the new firmware image. After performing an integrity check on the new binary image using a cryptographic key, the new firmware replaces the previous one, finalizing the update process. If the new image is corrupted or fails the check, the update is aborted and the old firmware is retained. This procedure is thoroughly described in [38]. Given a target application firmware size of approximately 100 KB, more than 900 fragments are necessary (considering 112 bytes per fragment, because the LoRaWAN protocol uses 3 bytes), resulting in a transmission duration that exceeds 100 min. The radio must remain continuously active during this period, significantly increasing energy consumption. Such prolonged activity conflicts with the system's intended operational design, which aims to stay in low-power mode most of the time, activating only for some seconds every few hours. Consequently, despite the infrequent nature of firmware updates, their associated energy costs remain excessively high.

To mitigate the inefficiency of full-image transmission, we integrated the proposed *bpatch* algorithm into the same architecture. Adopting the *bpatch* algorithm makes the delta file transmitted several times smaller than the entire firmware. Moreover, the existing communication protocol remains unaltered, with the only additional step being the patching operation to reconstruct the new firmware before rebooting. Fig. 1(b) provides an overview of the modified procedure. Instead of transmitting the entire new image, it is combined with the old version to generate a patch using *bpatch*. This patch is then transmitted to the device via the FUOTA protocol, where it is applied to reconstruct the new firmware. On the other hand, Fig. 2 provides details regarding the memory footprint of the improved system. Specifically, Fig. 2(a) illustrates the initial state of memory, Fig. 2(b) represents the memory state upon receiving the patch, Fig. 2(c) depicts the system state after reconstructing the updated firmware image, and Fig. 2(d) shows the final state where the new firmware replaces the previous version. As detailed in Section 4, *bpatch* introduces no complex computational requirements. This design ensures that the reconstruction process remains feasible on hardware-constrained MCUs without compromising the low-power objectives of the IoT node. The integration of *bpatch* into the MCU is straightforward and general, as it consists of a C function where only the memory read/write operations need to be implemented. This also allows the use of external memories, which is quite common in MCUs with limited flash capacity.

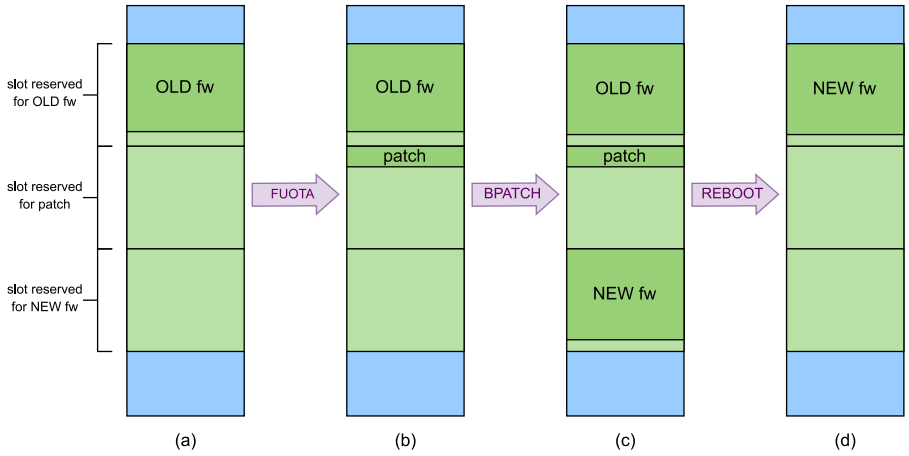

Fig. 2. Footprint of the MCU's flash. (a) initial state, (b) following FUOTA, (c) *bpatch* application, and (d) after a successful reboot.

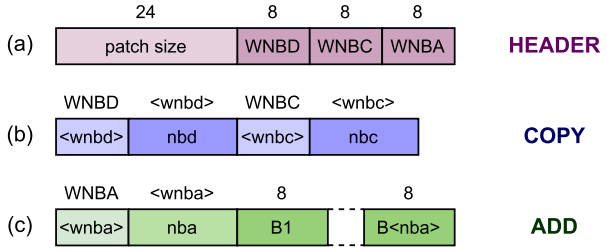

Fig. 3. Opcodes employed in *bpatch*: (a) header, (b) *COPY* and (c) *ADD*.

4. Description of *bpatch* algorithm

The desired characteristic of *bpatch* is to produce an extremely compact delta file that contains only simple instructions to construct the new file from the older one. It should be platform-independent and must be specific to binary files. In the following section, some considerations are exposed to illustrate the main idea of *bpatch*. Here, it is applied to LoRaWAN, as a case study, but the approach could be used for other protocols.

While binary firmware images inherently depend on specific hardware architectures, we notice that even minor code modifications tend to generate unpredictable and widespread differences. As investigated in [34], inserting or deleting functions or global variables can shift physical addresses, resulting in minor byte-level changes across the entire firmware image. Consequently, typical firmware modifications, such as adding a few functions or minor code adjustments, can produce small and pervasive differences in the entire firmware.

Considering this factor along with the requirements of *bpatch*, the proposed encoding strategy includes the following features:

- Only the two fundamental commands, *COPY* and *ADD*, are utilized.
- Byte positions in the delta file are expressed as incremental values rather than absolute positions, to reduce the range of numbers.
- Bit lengths allocated for command fields are dynamically customized rather than fixed.
- Each field is preceded by an indication of its bit length.
- No bits are wasted in indicating the command type (*COPY* or *ADD*), alternation maintaining the consistency.

The first phase of the algorithm consists of finding the difference between the old and new versions of the firmware. For this purpose, the UNIX diff program is exploited, and the binary files are converted into text files with one byte per row before applying the program. This choice is adopted because UNIX diff [39] is based on the Myers algorithm. This algorithm aims to find the shortest number of instructions that transform the first text file into the second one [40,41]. The instructions can be easily decomposed into *COPY* and *ADD* opcodes, which follow previously listed rules.

The innovation of this encoding lies in the optimal compression achieved by customizing bit allocation for opcode fields, thereby avoiding unnecessary bit wastage. The opcode formats are illustrated in Fig. 3, where each rectangle represents a field value, and the corresponding bit length is indicated above each field. Specifically, the *COPY* opcode, depicted in Fig. 3(b), includes two fields: *nbd* (number of bytes to discard in the original firmware) and *nbc* (number of bytes to copy from the original firmware). Conversely,

the *ADD* opcode, described in Fig. 3(c), includes a single field *nba*, representing the number of new bytes to insert, followed by the corresponding byte values. Bit lengths for each field are dynamically adjusted based on the values they must represent, and the bit length itself is indicated explicitly before each field. Consequently, smaller numerical values, prevalent in this case, consume fewer bits than fixed byte-lengths. The bit length of these pre-fields is fixed by constants (WNBD, WNBC and WNBA) established before generating the delta file, depending on the maximum of their values. These constants are contained in the patch inside a header, Fig. 3(a), along with the patch size expressed in *bits*.

Algorithm 1 bpatch reconstruction

```
patch_size ← READ(24)                          ▷ start header
WNBD ← READ(8)                                 ▷ read from patch 8 bits
WNBC ← READ(8)
WNBA ← READ(8)
i ← 0

while i < patch_size do
    wnbd ← READ(WNBD)                          ▷ start COPY opcode
    nbd ← READ(wnbd)
    JUMP(nbd)                                  ▷ skip nbd bytes from old
    i ← i + WNBD + wnbd                        ▷ update patch index
    wnbc ← READ(WNBC)
    nbc ← READ(wnbc)
    COPY(nbc)                                  ▷ copy nbc bytes from old to new
    i ← i + WNBC + wnbc
    wnba ← READ(WNBA)                          ▷ start ADD opcode
    nba ← READ(wnba)
    INSERT(nba)                                ▷ copy nba bytes from patch to new
    i ← i + WNBA + wnba + 8·nba
end while
```

An Application Programming Interface (API) is implemented in Python to handle *bpatch*. Firmware reconstruction is based on a dedicated C function, which loads the patch and original firmware into memory buffers and assembles the updated firmware according to patch instructions. This function can be integrated into an MCU to perform the FUOTA. The Python API and the C reconstruction software are openly available on GitHub https://github.com/vlsi-nanocomputing/bpatch and archived on Zenodo [9]. A concise summary of the reconstruction algorithm is provided in Algorithm 1. Initially, the patch size and the fixed field lengths are read from the header. Subsequently, an alternating sequence of *COPY* and *ADD* commands specifies the bytes to copy from the existing firmware version and the new bytes to insert, respectively. As previously noted, fields are not byte-aligned, and therefore, reading operations are performed at the bit level.

5. Comparison on delta algorithms

The compression performance of *bpatch* was assessed and compared with three differential open-source algorithms, *bsdiff*, *VCDIFF* and *HDiffPatch*, which are widely used for updating mobile applications [42] thanks to their ability to produce very compact patch files. However, it should be noted that smartphones and tablets offer orders of magnitude more memory, processing power, and energy than ultra-low-power MCUs typically found in IoT nodes. An algorithmic complexity acceptable in the mobile domain can become prohibitive on constrained devices. A brief overview of each algorithm is provided below:

- **bsdiff**: Utilizes a Suffix Array approach to identify differences, and it can recognize not only identical segments, but also partially similar segments by transmitting just the differing bytes. This algorithm typically generates large patch files filled with many zeros and repetitive sequences, which are subsequently compressed using *bzip2*, allowing for significantly reduced final patch sizes.
- **VCDIFF**: Employs a straightforward delta encoding approach based on byte-level *COPY* and *ADD* opcodes, with minimal complications. While the algorithm itself does not include compression, external compression is commonly applied to reduce the final patch size.
- **HDiffPatch**: Similar to *bsdiff* in principle, *HDiffPatch* provides considerably improved performance in terms of both computational efficiency and reduced patch size. As with *VCDIFF*, external compression can further decrease patch size but is not inherently integrated into the algorithm.

To assess performance, three Git repositories tracking firmware evolution for the selected MCU in three different real-world applications were analyzed. For consistency, compression strategies were turned off during testing; specifically, a variant of *bsdiff* without the *bzip2* compression stage was employed.

Fig. 4 compares patch sizes generated by each algorithm, along with the size of the new firmware image for comparison. Two variants of the *bpatch* are presented: the standard *"bpatch"*, which employs the basic UNIX diff command, and the enhanced *"bpatch_m"*, which incorporates the "minimal" option (available using option "-d"). This option applies an optimized variant of the

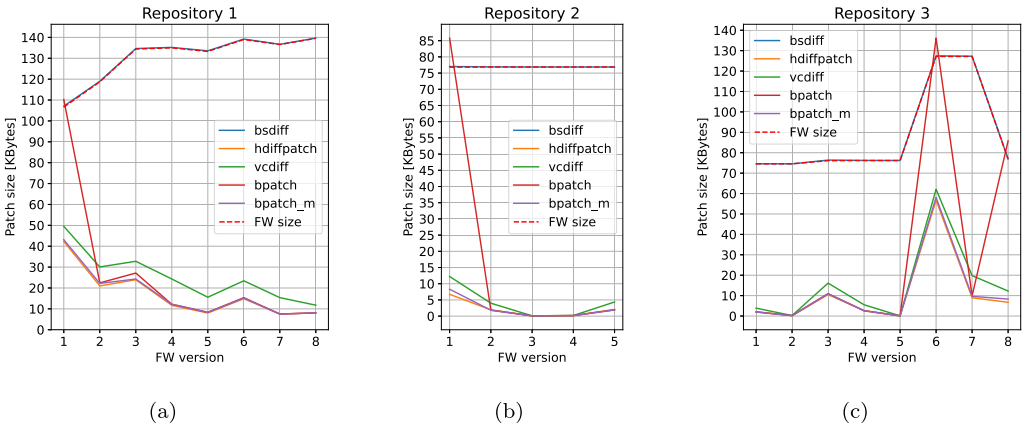

Fig. 4. Comparison of patch sizes produced by the five algorithms for the three Git repositories (a), (b), and (c).

diff algorithm to further compress the set of changes, potentially at the cost of increased processing time. Attempts to use alternative differential algorithms, such as the pure Myers algorithm described in [40] (UNIX diff strategies propose a slight variation of the Myers algorithm), did not yield significant improvements over *bpatch_m*, thus, they were excluded from the analysis.

From the three graphs, *HDiffPatch* and *bpatch_m* consistently generate the smallest patches, whereas *VCDIFF* produces slightly larger delta files, differing by only a few kilobytes. Conversely, *bsdiff* generates patches nearly equivalent in size to the original firmware, reflecting its dependence on external compression (*bzip2*) to achieve reduced patch sizes. The standard *bpatch*, utilizing the basic UNIX diff command, generally matches the performance of its enhanced counterpart, *bpatch_m*, except in certain cases where patch sizes are equal to or exceed the original firmware size.

To further analyze the hardware independence and general performance of *bpatch*, an additional evaluation was conducted using an open-source firmware repository containing incremental firmware versions. The ArduPilot website [43] provides numerous binary images used for drone boards. This source was chosen due to its variety of small-size binaries (under 2 MB), multiple supported boards, and numerous sequential firmware versions. Specifically, five projects were considered: *Plane*, *Copter*, *Rover*, *Sub*, and *Antenna Tracker*. Inside them, two folders were selected: *ACNS-CM4Pilot*, based on STM32H7 microcontrollers, and *navio*, based on Raspberry Pi boards. In total, 173 patches were generated from firmware images obtained from the three previous repositories and this website. Tests on larger binaries were not conducted for two main reasons. First, the proposed solution is designed explicitly for hardware-constrained MCUs, which typically do not handle large firmware images. Second, the UNIX diff, based on the Myers algorithm, is not optimized for large files with extensive differences. To provide clear insights, three firmware update scenarios were identified:

- **Major updates (MJ)**: Significant code refactoring, characterized by substantial code modifications and high byte-count variations.
- **Minor updates (MN)**: Only limited changes such as bug fixes, feature enhancements, or minor code adjustments. This scenario represents the primary target application for *bpatch*.
- **No updates/constant updates (NU)**: Situations where firmware versions remain essentially unchanged (e.g., documentation updates) or contain minimal byte differences (e.g. macro or constant modifications). While remote updates may be unnecessary in these cases, testing was conducted for completeness.

Patch sizes generated by *HDiffPatch* were used to categorize these scenarios: patches smaller than 0.5% of the total firmware size were classified as NU, whereas patches exceeding 20% were classified as MJ. *HDiffPatch* was selected as the reference due to its superior compression among the analyzed open-source tools.

Table 2 reports the *compression factor* defined as image size/patch size for every algorithm and platform. *bsdiff* was excluded from this comparison, as previously noted, it relies on an external *bzip2* pass for generating small patch files. The column labeled "STM32WL" corresponds to the Git repositories illustrated in Fig. 4, while the remaining two columns pertain to binaries from ArduPilot.

Both *bpatch* variants dominate the NU scenario. On STM32WL the factor reaches ~680×, versus 561× for *HDiffPatch* and 591× for *VCDIFF*. The gap widens on the larger STM32H7 and Raspberry Pi images, where *bpatch* attains 33k–39k× while the best competitor stays below 7k× . Hence, when only a handful of bytes change, as is common in IoT hot-fixes, *bpatch* delivers an order-of-magnitude better size reduction.

In the case of the minor updates (MN) scenario, *bpatch_m* equals or slightly surpasses *HDiffPatch* on every benchmark (18.0× vs. 17.8× on STM32WL, 15.1× vs. 15.5× on STM32H7, 9.37× vs. 10.13× on Raspberry Pi). The standard *bpatch* follows within a few percent, while *VCDIFF* lags.

When more than 20% of the firmware changes, the patch overhead becomes appreciable. The *bpatch* variant falls below unity on STM32WL (the patch is slightly larger than the new image) indicating that the basic UNIX diff algorithm is not sufficient to encode

Table 2
Average compression for the four algorithms for three different architectures.

	STM32WL			STM32H7			Raspberry Pi		
	MJ	MN	NU	MJ	MN	NU	MJ	MN	NU
bpatch	0.951	16.9	676	1.46	15.0	33 844	1.47	9.30	39 067
bpatch_m	2.33	18.0	676	3.31	15.1	33 844	2.34	9.37	39 067
hdiffpatch	2.38	17.8	561	4.04	15.5	6793	2.64	10.13	6955
vcdiff	2.1	9.5	591	2.06	4.3	14 396	1.64	3.43	13 568

Table 3
Current measurements for Class A and C in low-power mode.

	Avg current	Std current	Sample rate
Class A	1.75 μA	0.37 μA	1 k/s
Class C	6459.8 μA	7.5 μA	1 k/s

the differences effectively. However, the optimized *bpatch_m* recovers a 2.3–3.3× factor—essentially matching *HDiffPatch* (2.4–4.0×) and outperforming *VCDIFF* on two out of three platforms.

In summary, *bpatch_m* offers compression comparable to (and often better than) state-of-the-art algorithms across all scenarios, while standard *bpatch* excels whenever the modification is small. Conversely, both *HDiffPatch* and *VCDIFF* demand a filesystem and several resources to decompress patches, resources that are unavailable on many MCUs. By contrast, the constant-footprint reconstruction routine of *bpatch* runs entirely in place and is therefore immediately deployable on ultra-low-power IoT nodes.

6. Energy measurement and update time evaluation of *bpatch*

This section quantifies the energy and time savings obtained by *bpatch* in the beehive monitoring system introduced in Section 3. Competing delta tools are not considered here because, as discussed earlier, their reconstruction routines require a filesystem and RAM/CPU resources unavailable on the target MCU; a fair on-device comparison is therefore impossible. The test system is built around an STM32WL55JC microcontroller, equipped with 256 kB of on-chip flash and 64 kB of SRAM, and an external 8 Mbit MX25L8006EM1I-12G flash device.

Typically, the IoT node remains in sleep mode when not actively performing measurements, during which the MCU and peripherals are powered down, leaving only the Real-Time Clock (RTC) operational to wake up the MCU periodically. However, during FUOTA operations, the radio module must remain active to receive firmware fragments. LoRaWAN specifies three operational classes defining device behavior for downlink reception, which reflect the operating mode of the radio, the classes [44] are:

- **Class A**: After each uplink transmission, the device opens two brief receiving windows, thus limiting radio activation.
- **Class B**: Extends Class A functionality by periodically opening additional receiving windows (ping slots) synchronized via beacons transmitted by the server side
- **Class C**: The radio remains continuously active, always ready to receive downlink transmissions.

Under normal operating conditions, the device operates in Class A mode to minimize energy by deactivating the radio, but switches to Class C for a FUOTA session. We quantified the energy demand of Class A and C by measuring the current consumption when the device is not executing an operation and it is in sleep mode. Measurements were taken using a Tektronix DMM7510 digital multimeter, with the IoT device powered at 3.6 V with a power supply, corresponding to the nominal battery voltage. Current consumption was recorded at sample rates between 1 kHz and 100 kHz, depending on measurement duration.

Table 3 summarizes the current consumption comparison between the two operational classes over a 25 min duration, indicating average (Avg) and standard deviation (Std). Results demonstrate that Class C consumes significantly more energy, over three orders of magnitude greater than Class A. Consequently, reducing the FUOTA duration through fewer transmitted fragments directly translates into substantial battery energy savings.

A subsequent measurement aimed at estimating the current consumption per firmware fragment received was performed, thus establishing a baseline for the minimum energy required during FUOTA. We measured current consumption during the reception of 135 fragments and derived estimates for time and energy. The current samples of each fragment are distinguished from those collected during the device's sleep mode and are used to compute the total current and time. Fig. 5 provides a representative measurement for receiving one fragment. Initially, the device operates in Class C mode with the radio actively listening, consuming approximately 6.5 mA. Subsequently, the fragment is received, during which the current consumption slightly decreases to about 5.9 mA. Afterwards, the fragment is written into the MCU's memory, and finally, the node returns to Class C listening mode. The results, showing the average and standard deviation across the 135 fragments, are reported in Table 4.

Using these measurements, we estimated the total energy required for transmitting the full firmware update versus using incremental updates combined with *bpatch* (with the "minimal" option enabled). Energy consumption is expressed in *mAh* to assess battery impact intuitively. For the measurements, the firmware versions available in Repository 1 are taken as reference. Table 5 reports the comparative analysis, highlighting substantial improvements in time and energy when employing incremental updates.

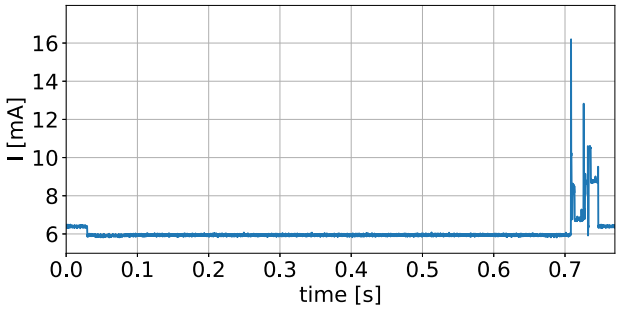

Fig. 5. Current measurements during the transmission of one fragment.

Table 4
Fragment current and time measurements.

	Avg	Std	Sample rate
Current	6060.58 μA	12 μA	8 k/s
Time	18.8 ms	3.3 ms	

Table 5
Energy and time reduction for update using *bpatch* on Repository 1.

Full-image update			Incremental update			Reduction	
Frags	Time [min]	Energy [mAh]	Frags	Time [min]	Energy [mAh]	Time [ratio]	Energy [mAh]
975	114	12.17	395	46	4.93	2.47	7.24
1086	127	13.55	203	24	2.53	5.35	11.02
1230	144	15.35	223	26	2.78	5.52	12.57
1235	144	15.41	112	13	1.40	11.03	14.02
1219	142	15.21	77	9	0.96	15.83	14.26
1271	148	15.86	139	16	1.73	9.14	14.13
1249	146	15.59	69	8	0.86	18.10	14.73
1276	149	15.93	74	9	0.92	17.24	15.01

Table 6
Energy and time estimations for *bpatch*.

Patch size [KB]	New FW size [KB]	Average current [mA]	Average time [ms]	Estimated energy [μAh]
22.44	118.72	12.14	4635.5	15.64
12.21	135.00	11.92	5028.2	16.65
8.35	133.28	11.89	4990.7	16.48
15.37	138.98	11.86	5117.1	16.85
7.50	136.58	11.86	5065.7	16.69

The table is organized as follows: for each case, the three columns under *Full-image update* and *Incremental update* report, respectively, the number of fragments (Frags), the transmission time, and the energy consumption when the update is performed with *bpatch*. The last two columns indicate the differences in these metrics between the two approaches. The benefits of incremental updates are evident, with transmission times reduced by up to 18× and proportional energy savings achieved.

It is important to note that these energy estimations consider only the fragment transmission phase, which represents the most prolonged and energy-intensive portion of the FUOTA process. They exclude the initial protocol setup and final device reboot phases, thus representing a conservative lower bound of total FUOTA energy consumption.

In the case of an incremental update, an additional computational overhead associated with firmware reconstruction must be considered. For this reason, we measured it on a selected set of firmware patches, labeled as MN. Results are presented in Table 6, the columns describe the firmware version selected, the average reconstruction time, and the associated energy consumption, averaged over five separate measurements. The reconstruction time is primarily determined by I/O operations, specifically reading the old firmware and the patch file, and subsequently writing the new firmware image. The energy required exhibits slight variations related to the new firmware size, which is considerably smaller, approximately three orders of magnitude lower than the energy consumption during the fragment transmission phase. Therefore, this additional reconstruction overhead can be considered negligible relative to the previously discussed energy savings.

The results show that *bpatch* achieves a consistent energy reduction compared to the standard FUOTA approach available in LoRaWAN, making it a valuable solution in scenarios where bandwidth and node hardware resources are limited. The energy savings

are proportional to the similarity between the two firmware versions, which strongly determines the patch size and, consequently, the amount of bandwidth saved.

7. Conclusions

This study presents a simple yet effective method for remotely updating IoT devices, achieving high compression of firmware images while ensuring minimal computational requirements during firmware reconstruction, thus making it feasible even for hardware-constrained microcontrollers and energy-constrained systems. The proposed solution is independent of the firmware binary structure and the architecture of the MCU. The differential algorithm leverages the robust and well-established UNIX diff software, enhanced by a carefully optimized representation of the edit script for effective compression.

The comparison between the open-source algorithms demonstrates that using a straightforward differential algorithm based on only two commands, combined with flexible bit-length encoding, is an efficient solution to reduce patch size without resorting to complex approaches. The extended number and variability of binary pairs enable a fair comparison and assessment of *bpatch* against other algorithms across several real-case scenarios. The advantage in time and energy for updates categorized as "minor updates" or "no updates" is undeniable, ranging from 9–18× and 500× up to 34k×, respectively. Even in the case of "major updates", the advantage is less pronounced. However, since the radio is one of the most energy-consuming components in the IoT system under investigation, even a 2× reduction in data size can significantly extend battery lifetime. The integration of *bpatch* into an ultra-low-power IoT system demonstrates its ease of deployment within both the communication protocol (LoRaWAN in our case) and the resource-constrained MCU. The measurements clearly show substantial energy savings, which is often the most critical factor in such applications, as it directly translates into a longer operational lifetime without manual intervention. In conclusion, *bpatch* represents an effective and energy-aware solution for FUOTA, significantly reducing the impact on battery lifespan.

Future work will explore editing script structure enhancements and possibly introducing additional or specialized opcodes to achieve even better compression. A key challenge will be maintaining simplicity in both structure and firmware reconstruction processes. In addition, future developments will focus on integrating multicast FUOTA into the proposed system. Such an extension would enable firmware dissemination, thereby improving scalability and enhancing the approach's applicability to large-scale IoT networks. Finally, while data compression programs commonly utilized by other differential algorithms have not been evaluated in this study, due to the primary objective of minimizing computational overhead at the edge, their integration could benefit more capable hardware platforms to reduce patch sizes further and optimize data transmission.

CRediT authorship contribution statement

Andrea De Simone: Writing – original draft, Validation, Software, Methodology, Conceptualization. **Giovanna Turvani:** Writing – review & editing. **Fabrizio Riente:** Writing – review & editing, Writing – original draft, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data and source code are available on github and zenodo as reported in Ref. [9].

References

[1] M. Centenaro, L. Vangelista, A. Zanella, M. Zorzi, Long-range communications in unlicensed bands: the rising stars in the iot and smart city scenarios, IEEE Wirel. Commun. 23 (5) (2016) 60–67, http://dx.doi.org/10.1109/MWC.2016.7721743.

[2] J. Catalano, Lorawan firmware update over-the-air (FUOTA), J. ICT Stand. 9 (1) (2021) 21–34, https://journals.riverpublishers.com/index.php/JICTS/article/view/7017.

[3] W. Mao, Z. Zhao, Z. Chang, G. Min, W. Gao, Energy-efficient industrial internet of things: Overview and open issues, IEEE Trans. Ind. Informatics 17 (11) (2021) 7225–7237.

[4] M. Pule, A. Yahya, J. Chuma, B. Letswamotse, Firmware updates over the air mechanisms for low Power Wide Area networks: A review, in: Proceedings of the 2019 International Conference on Wireless Networks and Mobile Communications, WINCOM, 2019, pp. 1–8, http://dx.doi.org/10.1109/WINCOM47513.2019.8942505, https://www.researchgate.net/publication/339591929_Firmware_Updates_Over_the_Air_Mechanisms_for_Low_Power_Wide_Area_Networks_A_Review.

[5] B. Vejlgaard, M. Lauridsen, H.C. Nguyen, B. Vojcic, P. Mogensen, M. Sorensen, Energy consumption model for sensor nodes based on LoRa and lorawan, in: Proceedings of the IEEE 85th Vehicular Technology Conference (VTC Spring), 2017, pp. 1–5, http://dx.doi.org/10.1109/VTCSpring.2017.8108182, https://ieeexplore.ieee.org/document/8108182.

[6] N. Chollet, N. Bouchemal, A. Ramdane-Cherif, Energy-efficient firmware over-the-air update for tinyml models in LoRaWAN agricultural networks, in: 2022 32nd International Telecommunication Networks and Applications Conference, ITNAC, 2022, pp. 1–6, http://dx.doi.org/10.1109/ITNAC53133.2022.9977800, https://www.computer.org/csdl/proceedings-article/itnac/2022/09998338/1JGZVzIbOa4.

[7] K. Mikhaylov, J. Petaejaejaervi, T. Haenninen, Comparison of LoRaWAN classes and their power consumption, in: Proceedings of the 13th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, ISSNIP, 2017, pp. 1–6, http://dx.doi.org/10.1109/ISSNIP.2017.8007361, https://ieeexplore.ieee.org/document/8007361.

[8] L. Casals, B. Mir, R. Vidal, C. Gomez, Modeling the energy performance of lorawan, Sensors 17 (10) (2017) 2364, http://dx.doi.org/10.3390/s17102364, https://www.mdpi.com/1424-8220/17/10/2364.

[9] De Simone Andrea, Fabrizio Riente, Giovanna Turvani, Vlsi-nanocomputing/bpatch: version-1.0.0, Zenodo, 2025, http://dx.doi.org/10.5281/zenodo.15348169.

[10] C. Percival, Binary diff/patch utility, 2025, https://www.daemonology.net/bsdiff/. (Accessed 19 March 2025).

[11] google, Open-vcdiff, 2025, https://github.com/google/open-vcdiff. (Accessed 19 March 2025).

[12] sisong, HDiffPatch, 2025, https://github.com/sisong/HDiffPatch. (Accessed 19 March 2025).

[13] S. Brown, C.J. Sreenan, Software updating in wireless sensor networks: A survey and lacunae, J. Sens. Actuator Networks 2 (4) (2013) 717–760.

[14] M.J.B. de Sousa, L.F.G. Gonzalez, E.M. Ferdinando, J.F. Borin, Over-the-air firmware update for iot devices on the wild, Internet Things 19 (2022) 100578.

[15] Q. Wang, Y. Zhu, L. Cheng, Reprogramming wireless sensor networks: challenges and approaches, IEEE Netw. 20 (3) (2006) 48–55.

[16] J.L. Hernández-Ramos, G. Baldini, S.N. Matheu, A. Skarmeta, Global internet of things summit (gIoTs), IEEE 2020 (2020) 1–5.

[17] M.M. Villegas, C. Orellana, H. Astudillo, A study of over-the-air (ota) update systems for cps and iot operating systems, in: Proceedings of the 13th European Conference on Software Architecture, vol. 2, 2019, pp. 269–272.

[18] M. Rizzi, P. Ferrari, A. Flammini, E. Sisinni, Evaluation of the iot lorawan solution for distributed measurement applications, IEEE Trans. Instrum. Meas. 66 (12) (2017) 3340–3349.

[19] K. Mikhaylov, J. Petaejaejaervi, T. Haenninen, Analysis of capacity and scalability of the lora low power wide area network technology, in: European Wireless 2016; 22th European Wireless Conference, VDE, 2016, pp. 1–6.

[20] F. Van den Abeele, J. Haxhibeqiri, I. Moerman, J. Hoebeke, Scalability analysis of large-scale lorawan networks in ns-3, IEEE Internet Things J. 4 (6) (2017) 2186–2198.

[21] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, A. Fragkiadakis, Firmware over-the-air programming techniques for iot networks-a survey, ACM Comput. Surv. 54 (9) (2021) 1–36.

[22] W. Mao, Z. Zhang, Y. Li, W. Xu, T. Gu, J. He, Reliable and energy-efficient reprogramming for smart lorawan, in: 2023 IEEE Smart World Congress, SWC, 2023, pp. 1–8, http://dx.doi.org/10.1109/SWC57546.2023.10449002, https://www.researchgate.net/publication/378668698_Reliable_and_Energy-Efficient_Reprogramming_for_Smart_LoRaWAN.

[23] V. Malumbres, J. Saldana, G. Berné, J. Modrego, Firmware updates over the air via lora: Unicast and broadcast combination for boosting update speed, Sensors 24 (7) (2024) 2104.

[24] D. Heeger, M. Garigan, E. Eleni Tsiropoulou, J. Plusquellic, Secure lora firmware update with adaptive data rate techniques, Sensors 21 (7) (2021) 2384.

[25] Z. Sun, T. Ni, H. Yang, K. Liu, Y. Zhang, T. Gu, W. Xu, Flora: Energy-efficient, reliable, and beamforming-assisted over-the-air firmware update in LoRa networks, in: Proceedings of the 22nd International Conference on Information Processing in Sensor Networks, IPSN, 2023, pp. 14–26, http://dx.doi.org/10.1145/3583120.3586963, https://dl.acm.org/doi/10.1145/3583120.3586963.

[26] A. Anastasiou, P. Christodoulou, K. Christodoulou, V. Vassiliou, Z. Zinonos, Iot device firmware update over lora: The blockchain solution, in: 2020 16th International Conference on Distributed Computing in Sensor Systems, DCOSS, IEEE, 2020, pp. 404–411.

[27] C.-Y. Park, S.-J. Lee, I.-G. Lee, Secure and lightweight firmware over-the-air update mechanism for internet of things, Electronics 14 (8) (2025) 1583.

[28] Z. Sun, T. Ni, H. Yang, K. Liu, Y. Zhang, T. Gu, W. Xu, Flora+:. Energy-efficient, reliable, beamforming assisted, And secure over-the-air firmware update in lora networks, ACM Trans. Sens. Networks 20 (3) (2024) 1–28.

[29] W. Wei, J. Banerjee, S. Islam, C. Pan, M. Xie, Energy-aware incremental OTA update for flash-based batteryless IoT devices, in: 2024 IEEE Computer Society Annual Symposium on VLSI, ISVLSI, 2024, pp. 51–56, http://dx.doi.org/10.1109/ISVLSI56778.2024.00018, https://arxiv.org/abs/2406.12189.

[30] H.D. Nguyen, N. Le Sommer, Y. Mahéo, Over-the-air firmware update in lorawan networks: A new module-based approach, Procedia Comput. Sci. 241 (2024) 154–161.

[31] L. Pereira, L. Oliveira, A. Rodrigues, Efficient runtime firmware update mechanism for LoRaWAN class a devices, Eng 5 (4) (2024) 2610–2632, http://dx.doi.org/10.3390/eng5040137, https://www.mdpi.com/2673-4117/5/4/137.

[32] W. Wei, S. Islam, J. Banerjee, S. Zhou, C. Pan, C. Ding, M. Xie, Intermittent OTA code update framework for tiny energy harvesting devices, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 44 (1) (2025) 77–90, http://dx.doi.org/10.1109/TCAD.2024.3418396, https://dl.acm.org/doi/abs/10.1109/TCAD.2024.3418396.

[33] O. Kachman, M. Baláz, Effective over-the-air reprogramming for low-power devices in cyber-physical systems, in: Doctoral Conference on Computing, Electrical and Industrial Systems (DoCEIS), in: IFIP Advances in Information and Communication Technology, vol. 470, 2016, pp. 284–292, http://dx.doi.org/10.1007/978-3-319-31165-4_28, https://link.springer.com/chapter/10.1007/978-3-319-31165-4_28.

[34] T. Stathopoulos, J. Heidemann, D. Estrin, Efficient code distribution in wireless sensor networks, in: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04), 2004, pp. 282–293, http://dx.doi.org/10.1145/1031495.1031526, https://dl.acm.org/doi/10.1145/1031495.1031526.

[35] O. Kachman, M. Baláz, Reprogramming with minimal transferred data on wireless sensor network, in: Doctoral Conference on Computing, Electrical and Industrial Systems (DoCEIS), in: IFIP Advances in Information and Communication Technology, vol. 470, 2016, pp. 284–292, http://dx.doi.org/10.1007/978-3-319-31165-4_28, https://link.springer.com/chapter/10.1007/978-3-319-31165-4_28.

[36] A. De Simone, L. Barbisan, G. Turvani, F. Riente, Advancing beekeeping: Iot and tinyml for queen bee monitoring using audio signals, in: IEEE Transactions on Instrumentation and Measurement, 2024.

[37] STMicroelectronics, AN5554, 2025, https://www.st.com/resource/en/application_note/an5554-lorawan-firmware-update-over-the-air-with-stm32cubewl-stmicroelectronics.pdf. (Accessed 20 March 2025).

[38] STMicroelectronics, AN5056, 2025, https://www.st.com/resource/en/application_note/an5056-integration-guide-for-the-xcubesbsfu-stm32cube-expansion-package-stmicroelectronics.pdf. (Accessed 21 March 2025).

[39] Gnu, Diffutils, 2025, https://www.gnu.org/software/diffutils/. (Accessed 7 April 2025).

[40] W. Miller, E.W. Myers, A file comparison program, Softw.: Pr. Exp. 15 (11) (1985) 1025–1040.

[41] E.W. Myers, An o (nd) difference algorithm and its variations, Algorithmica 1 (1) (1986) 251–266.

[42] Z. Sun, S. Jiang, K. Liu, T. Gu, J. He, Understanding differencing algorithms for mobile application updates, IEEE Trans. Mob. Comput. 23 (12) (2024) 1234–1245, http://dx.doi.org/10.1109/TMC.2024.3407867, https://www.computer.org/csdl/journal/tm/2024/12/10543054/1XorhOO1Ebu.

[43] ArduPilot, ArduPilot firmware download, 2025, https://firmware.ardupilot.org/. (Accessed 25 June 2025).

[44] T.T. Network, Device classes, 2025, https://www.thethingsnetwork.org/docs/lorawan/classes/. (Accessed 6 May 2025).