Firmware Over-the-air Programming Techniques for IoT Networks - A Survey

KONSTANTINOS ARAKADAKIS, PAVLOS CHARALAMPIDIS,
ANTONIS MAKROGIANNAKIS, and ALEXANDROS FRAGKIADAKIS,
Institute of Computer Science, Foundation for Research and Technology-Hellas, Greece

The devices forming Internet of Things (IoT) networks need to be re-programmed over the air, so that new features are added, software bugs or security vulnerabilities are resolved, and their applications can be re-purposed. The limitations of IoT devices, such as installation in locations with limited physical access, resource-constrained nature, large scale, and high heterogeneity, should be taken into consideration for designing an efficient and reliable pipeline for over-the-air programming (OTAP). In this work, we present a survey of OTAP techniques, which can be applied to IoT networks. We highlight the main challenges and limitations of OTAP for IoT devices and analyze the essential steps of the firmware update process, along with different approaches and techniques that implement them. In addition, we discuss schemes that focus on securing the OTAP process. Finally, we present a collection of state-of-the-art open-source and commercial platforms that integrate secure and reliable OTAP.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Networks → Network protocols**; • **Security and privacy**;

Additional Key Words and Phrases: Internet-of-Things (IoT), over-the-air-programming, firmware update, code dissemination, delta scripts, firmware image similarity

ACM Reference format:
Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. 2021. Firmware Over-the-air Programming Techniques for IoT Networks - A Survey. *ACM Comput. Surv.* 54, 9, Article 178 (October 2021), 36 pages.
https://doi.org/10.1145/3472292

Konstantinos Arakadakis also with Department of Computer Science, University of Crete.
This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-03389).
Authors' address: K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, Institute of Computer Science, Foundation for Research and Technology-Hellas, 100 Plastira Str, Heraklion, 70013, Greece; emails: konarak@csd.uoc.gr, {pcharala, makrog, alfrag}@ics.forth.gr.

0360-0300/2021/10-ART178 $15.00
https://doi.org/10.1145/3472292

1 INTRODUCTION

The **Internet of Things (IoT)** presents itself as an emerging technology that is able to interconnect a massive number of heterogeneous smart devices for supporting complex data-driven applications in a variety of domains, such as smart cities, healthcare, industrial automation, and so forth. The advances in micro-electromechanical systems that enabled the development of low-cost sensors, the progress of wireless communications in the field of **Wireless Sensor Networks (WSNs)**, and the growing market demand for machine-to-machine interaction have been identified as some of the key factors that led to the remarkable popularity and adoption of the IoT technology [69]. It is common for the devices forming an IoT network to operate unattended for long periods in variable environmental conditions. Irrespective of the care taken during the development phase, the IoT devices need frequently to be re-programmed **over the air (OTA)**, either for resolving bugs or security vulnerabilities (identified after deployment) or supporting different features and/or applications. Failing to do so may result in decreased network performance and security breaches that could compromise the privacy and safety of users, and in general undermine the long-term sustainability of the IoT deployment. However, the dynamic, heterogeneous, and resource-constrained nature of IoT networks should be taken carefully into consideration for achieving a dependable and efficient OTA programming.

A fundamental characteristic of the IoT networks is the dynamic changes of the network topology that can happen because of either the energy depletion of nodes or their inability to communicate with adjacent (neighboring) nodes. Furthermore, very often, IoT devices are equipped with scarce resources, such as limited memory, storage, and processing power to keep the cost and battery consumption low. These characteristics impose additional challenges on both the firmware design the nodes run and their update during the lifetime of the network. Regardless of the attention given during the development period, software bugs can occur at any level of the system and stage of the development cycle. As stated in [32], an unexpected combination of inputs that are received by the nodes of a network can stimulate untested firmware branches, resulting in unresponsive nodes that may degrade Quality of Service or even the integrity of the network. Hence, firmware updates are often released to fix such bugs and security breaches, introduce new functionality, or even change the purpose of the application completely. The latter type of update is a common practice when the behavior of the device needs to be altered dynamically on par to changes of the environment and the available storage is too restricted to accommodate multiple application images. In the most common scenario, however, the updates introduce minimal modifications to the firmware code, changing the implementation of a few functions or reconfiguring application parameters [91].

Traditionally, in order to update the nodes of a WSN (being commonly at the core of an IoT system), maintenance personnel had to be dispatched and access the nodes via a serial port or other hardwired back channel. The problem with this solution is that it is not scalable and requires a vast amount of time, which may be intolerable when the update includes fixes for security breaches and should be installed by the nodes as soon as possible. Furthermore, the physical access to the nodes may be impossible sometimes, since they can be located in inaccessible areas (e.g., implanted into asphalt roads) or implanted into human bodies as medical sensors [104].

Due to the above limitations, in the early 2000s the first wireless update schemes for restricted embedded devices were developed. These schemes required the firmware image to be built at a base station that would afterward transmit it in its entirety to the neighboring nodes over a radio channel. Once a node had fully received the update, it rebooted and the bootloader overwrote the contents of the internal flash memory (program memory) with the new image and started running the new firmware. A limitation of these first wireless solutions [98], however, is that only the nodes

that were within the radio range of the base station (over a single hop) were able to receive the update. Nowadays, due to the large geographical scale of modern IoT networks, multi-hop communication is a reality, and in order to facilitate proper OTAP, new *dissemination protocols* have been designed, whose goal is the reduction of the energy consumption upon an update, avoiding redundant transmissions and collisions that can degrade the quality of the wireless channel. In addition, as the developers used to update the firmware frequently, the approach of transmitting the entire firmware image each time a new update had been released quickly became obsolete, as it seriously affected the lifetime of these devices, which were supposed to run even for years. This limitation, accompanied by the vast amount of time the update of a network required [95], triggered the development of the first *incremental programming* schemes. These schemes avoid sending the whole firmware image every time a new update is released and just transmit commands to the nodes that instruct them how to reconstruct the new firmware locally, utilizing parts of the currently run firmware that each node has already stored in its flash memory. Thus, in order to update the modern IoT networks incrementally, a base station should first create the new firmware image and then the resulting delta script (patch), computing the common segments between the new and the current firmware versions. Afterward, the delta script is disseminated in the network, utilizing a multi-hop protocol, in order to reach all nodes, whereupon each node should interpret the received script, execute the commands found inside, and reconstruct the new firmware locally. Once this last step has been completed, the node can be updated, replacing the firmware it currently runs with the one it just reconstructed (loading phase). This process can be visualized in Figure 1.

The importance of the security during the update of the IoT devices must also be noted. Since most of the protocols that have been designed for this environment aim for the update image to reach a lot of the nodes of the network (if not all of them), they follow a propagation approach where each node just needs to receive a part of the update and it forwards it to its neighbors. Although this technique aids the fast update of the network, it raises serious concerns regarding the security of the process. If nodes do not validate the authenticity of the updates they receive, this could result in the installation of faulty or malicious firmware in the nodes, originating from either outsiders or already compromised network devices. However, when a new update scheme is designed, the limitations that bind these devices [71], due to their restricted nature, should be taken into account. For example, actuators and sensors are devices with low-power antennas and limited transmission range, able to reach only a portion of the other nodes in the network. Furthermore, these devices are severely constrained in terms of computational power, memory, and storage [19]. These characteristics dramatically restrict the features that can be embedded into the firmware image and the update mechanism itself [16]. In general, IoT OTAP is not a trivial task and poses many challenges that can affect the quality and sustainability of the network. The main challenges and limitations that complicate the OTAP process and have been reported in the literature are presented in Section 2.

In this survey, we examine the main stages of the wireless update process, mainly focusing on the severely restricted IoT devices. For each such stage, we also present major contributions that have been designed for this class of devices, as well as the limitations and complications the researchers had to overcome. We advocate that the division of the OTAP process we have followed (found in Figure 1) is the most suitable, as it presents the reasonable flow for updating the nodes in an incremental fashion.

Most surveys usually target a specific aspect of the OTAP process, neglecting the other stages and their interconnection. For example, the authors of [86] divide the update process in a similar way to us but specifically target the dissemination stage of the update, neglecting the delta generation and the security aspects of the process. In [20], the authors present an extensive overview of many update dissemination protocols for WSNs, as well as stand-alone update schemes. However,

they avoid presenting the internals of various schemes in higher detail and they compare the protocols and schemes using the same metrics. We believe that such a comparison is not appropriate because most contributions introduce novelties, concentrating on different aspects of the update process, and they are often supposed to be used as parts of an update scheme. Thus, a more careful division and comparison of them are required. On the other hand, in [106] and [103], the security aspects of the update process are discussed, providing valuable information on how an adversary can exploit the epidemic nature of the dissemination protocols to initiate attacks (e.g., **Denial-of-Service (DoS)**, install malicious code, etc.). Moreover, the authors provide information about the available cryptographic libraries that are suitable for constrained devices, as well as their memory footprint and performance. However, these surveys do not present any actual contributions that have utilized these libraries to ensure the authenticity and the integrity of the transmitted data during the update. In [22], a lot of security-oriented dissemination protocols are presented along with some authentication and freshness verification methods. Finally, the authors in [18] focus on the key principles of OTAP in IoT networks; however, they limit their contribution in brief description of these principles, without presenting and analyzing specific research contributions in depth.

Although these surveys provide valuable information for the insights of the wireless update process in IoT networks, our survey differs in several aspects. The main contributions of this article focus on OTAP techniques, proposed over the period 1999–2020, and are as follows: (1) we perform a comprehensive organization of the OTAP process; (2) we highlight the insights and challenges of each step, as well as analyze extensively and compare contributions related to each step; (3) we discuss the limitations of the restricted embedded devices that must be taken into account when designing a new update scheme; (4) we highlight the security aspects of the update process and present contributions that ensure the integrity and authenticity of the received firmware binary during the dissemination stage; and (5) we present and compare some state-of-the-art open source and commercial IoT platforms that integrate secure and reliable OTAP support.

2 MAIN CHALLENGES AND LIMITATIONS FOR OVER-THE-AIR PROGRAMMING

2.1 Limited Memory, Storage, and Processing Power

Usually, IoT nodes are restricted embedded devices with limited memory and storage size. Such nodes typically feature a relatively low-cost internal NAND-based flash memory [89], called *program flash* that accommodates both the bootloader (piece of software responsible for writing the new firmware code in memory) and the firmware code. Moreover, there is an SRAM or DRAM for the storage of the volatile data, including the heap, the stack, and the global variables of the application. These sections are mapped to predetermined RAM regions when the *reset handler* executes [105], a purpose-specific code, whose goal is to prepare the RAM and initialize the registers before the actual application code starts running. Very often, the available RAM in IoT devices is too limited to accommodate the firmware code, so the flash memory is used for this purpose instead. Furthermore, IoT nodes are often equipped with an external non-volatile EEPROM for the storage of various data, such as routing tables and other network-related data. Moreover, many wireless update schemes use the EEPROMs for the storage of rollback images (golden image) to provide fail-safe updates. Additionally, EEPROMs are also used by many OTAP schemes [78, 79, 90] as a temporary storage for the update image. In these schemes, when the image has been received completely, the node copies the new firmware code from EEPROM to the program flash and then starts executing the new firmware. This strategy allows nodes to remain operational while receiving the new firmware image. On the other hand, other schemes (e.g., [41]) explicitly use the program flash to store the new firmware along with the currently running one (the flash memory

is divided into two equally sized regions). Finally, due to their low cost, IoT nodes typically feature micro-controllers that operate at a lower frequency compared to traditional CPUs [38]. Thus, if such a node executes a complex algorithm that needs intensive processing, the time overhead will be significantly high, and the node may be unable to perform any other operations. To address this limitation, the research community works toward lightweight algorithms in software (e.g., [74]), as well as hardware-based acceleration of cryptographic operations (e.g., [17]).

2.2 Flash Memory Degradation

Another challenge for the firmware update process is the quality degradation of the flash memory, caused by the large number of erase and write operations during an update. Flash memories consist of erase units called *blocks*. Prior to writing into a specific block, this must be erased; however, the number of times a block can be erased is limited. Typically, NAND-based flash memories offer 1 million flash cycles maximum, 10 times the life of a NOR-based memory [97]. When the erase threshold of a block has been reached, it is marked as a *bad block* and cannot be used in the future, thus limiting the available storage. In order to prolong the lifetime of flash memory, blocks must be utilized with caution when a new firmware image has to be stored, so one can achieve a uniform and smooth degradation of the available blocks. Some schemes (e.g., [30, 56, 62, 94]) exclusively use the RAM for storing temporal segments of a firmware image, avoiding redundant access and unnecessary operations on the flash memory.

2.3 Energy Consumption

IoT nodes are often battery operated, and thus constrained in terms of energy, while operating unattended in harsh environments for long periods of time [37]. Furthermore, they often rely on ambient power sources such as solar energy, wireless energy, and RF to operate uninterrupted. A node's lifetime is highly affected by routine operations such as those involved in the wireless radio communication. As an example, the transmission of a single bit of data could consume roughly the same amount of energy as executing 1,000 instructions [87]. Writing data into the flash memory is not a lightweight process either, as a higher voltage is required; hence, the total power consumption is significantly affected by the size of the data to be stored, and for this reason, firmware size minimization is of utmost importance. Furthermore, the update process can be energy intensive due to the amount of messages that need to be transmitted in the network, so firmware dissemination protocols must be carefully implemented to avoid redundant transmissions (e.g., flooding). Moreover, firmware decompression at the receiving node can consume a significant amount of energy, hence rendering compression-based update strategies unsuitable for use by constrained IoT nodes. Finally, nodes' energy consumption can be affected by the firmware code layout in the flash memory. In [77], the authors show that crossing a page boundary in flash memory causes additional energy consumption due to the extra circuitry powered up to read the new page. This implies that the loops in the firmware code must be properly aligned in a page, if possible, to avoid an excessive energy consumption.

2.4 Overhead Due to Node Reboot

In several OTAP contributions (e.g., [29, 31, 78, 79]), after the installation of a new firmware version has been completed, the node is required to reboot, so the bootloader is run and the new firmware starts executing. This can introduce a significant overhead in time-critical IoT applications such as those used in aviation, healthcare, connected cars, and so forth. Several other OTAP contributions (e.g., [107]) address this issue by proposing mechanisms able to dynamically patch an IoT node with a new firmware while it is operating, without the need to reboot.

2.5 Group-wise IoT Node Re-programming in Heterogeneous Environments

Another challenge for the firmware update process is that IoT nodes within a network can execute different firmware versions, have different roles, and hence have different update requirements. Moreover, these devices can be heterogeneous in terms of software and hardware [38, 45]. Under these conditions, the update mechanism must have the means to support the update of individual nodes in case of different firmware versions, and simultaneous updating in case of a group of nodes that have a common version. Group programming can minimize the energy consumption and the total re-programming time required. However, this is not trivial task as an efficient dissemination algorithm should be able to select optimum routing paths, in order to disseminate a new firmware version to a number of nodes. To make this feasible, modification of routing protocols may be required (e.g., [14]). In some contributions (e.g., [108]), the nodes can make decisions based on criteria like the current firmware version, the number of reachable neighbors, and so forth, to decide if they will accept the update. Most contributions, however, follow a two-stage approach to provide group-wise updates. Initially, metadata-carrying packets are flooded in the network and interested receiving nodes respond back. Following this approach, a routing tree is established and the nodes along this path act as firmware forwarding nodes [86], and the transmission of the actual firmware then starts. Finally, it must be noted that the update methods in such heterogeneous environments have to be portable across the hardware platforms and operating systems that nodes are based on, avoiding dependencies on libraries (e.g., cryptographic libraries [40, 63]) and underlying protocols.

2.6 Network Flooding

IoT nodes often communicate in low-bandwidth channels and in frequency bands that are overcrowded (i.e., in the unlicensed bands of 2.4 GHz) with significant interference. Moreover, the nodes can be dispersed in large geographical areas, for example, in IoT-enabled smart city applications [99]. OTAP protocols should avoid flooding the network during the update process by employing techniques for: (1) efficient selection of the nodes to be updated, (2) firmware compression, (3) dynamic patching, and (4) energy-aware routing for path selection.

2.7 Data and State Consistency

There are various algorithms (e.g., trust-based schemes [39], routing algorithms [68], etc.) where nodes cooperatively report measurements, so a central server can infer about a possible event. During the firmware update process, these applications can be severely disrupted, as some nodes may execute the new firmware version, while others the current one. For this reason, extra care should be taken during the update process and a suitable mechanism has to be used to guarantee data and state consistency. Usually, after an update, nodes reboot, so data and state are reset. Moreover, IoT networks are susceptible to changes of the environmental conditions [96] that can result in temporary node disconnections.

2.8 Security

IoT proliferation has significantly extended the attack surface, and many attacks with disastrous results have already taken place (e.g., Mirai and other botnets [13, 58]). Having compromised thousands of IoT devices around the world, an attacker can demonstrate large-scale DDoS attacks against critical infrastructures; hence, it is of paramount importance that IoT nodes are supported by an OTAP mechanism for securing bug fixes. Moreover, the OTAP mechanism itself has to be securely designed; otherwise, it can become an additional attack vector for an adversary. For

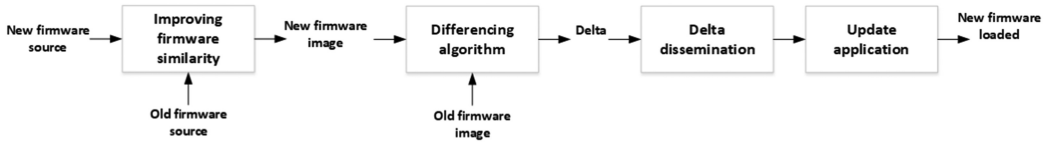

Fig. 1. Firmware update process essential stages [56].

example, the Zigbee Worm [88] was able to trigger a chain reaction of infections, initialized by a single compromised IoT device (light bulb), using a malicious firmware update image.

3 OVER-THE-AIR PROGRAMMING ESSENTIAL OPERATIONS

The limitations imposed by the inherent nature of IoT networks, as described in Section 2, dictate the careful design of an OTAP pipeline that mainly focuses on firmware image size reduction, as well as efficient and robust dissemination in the network. The essential stages for performing a firmware update, which are illustrated in the form of a flow-graph in Figure 1, are [56]:

- **Firmware similarity improvement**, which includes comparing the new firmware source code with the current one, in order to mitigate function or variable shifts and increase the similarity between the built firmware versions
- **Differencing algorithm application**, which includes producing a so-called *delta script* by using a differencing algorithm for comparing the current firmware image with the new one. The delta script encodes a set of instructions that, once applied on the current firmware, enable the reconstruction of the new one. In general, a delta script should be of minimal size (smaller than the original firmware image), since the goal is to reduce the data transmitted to the node
- **Delta script dissemination**, which is responsible for orchestrating the efficient and reliable firmware (delta script) distribution to the IoT nodes by applying a suitable dissemination protocol that focuses on transmitted data minimization
- **Update application**, which refers to the OTAP stage that takes place on the node and includes reconstructing, verifying, installing, and executing the new firmware

This section provides a detailed analysis of techniques related to all four stages of the OTAP pipeline and highlights both their advantages and limitations.

3.1 Improving Firmware Images Similarity

Despite the fact that a firmware update can introduce small modifications to the firmware code, these changes can result in a disproportional increase in the size of the delta script produced. As an example, suppose that a software developer modifies a single function within a piece of software that will be executed in an IoT node. As a result, the following instructions will be inserted in the delta script for transmission: ***COPY*** the firmware image segment from address 0x00000000 until the address where the function is located; ***ADD*** the segment that contains the new function implementation from the new firmware image; ***COPY*** the firmware image segment from the address following the end of the function implementation, until the end of the current firmware. Although this sequence of operations is correct, it constitutes a poor representation of the resulting delta script because in the binary image of the updated firmware, the code that follows the modified function's implementation will be relocated (shifted), to comply with the new size of the updated function; thus, functions that are located after the modified one will be shifted to other memory addresses (different from the ones in current version). Subsequently, the instructions that call these functions will use different target addresses (the address of the called function in the flash memory) in the two

versions. Finally, since all these modified target addresses will be encoded using *ADD* instructions in the script, the size of the generated delta script will be significantly high. This is also true when a new global variable is defined or a previously defined one is removed, as other global variables may need to relocate and hence the instructions that reference them will have different target addresses.

In [90], the authors distinguish four distinct properties of the firmware image that affect the size of the generated delta script.

- **Function shifts**. When a function is modified, either shrunk or grown, other functions located in lower memory addresses are forced to relocate; hence, the target addresses of the corresponding calling instructions need to change. In this case, a *COPY* instruction will be injected into the delta script to encode the common segments present in the two firmware versions, and an *ADD* instruction will be required to encode the bytes associated with the target addresses.
- **Global variables shifts**. The insertion of new global variables within the code can also affect the structure of the resulted binary image. Global variables are stored in RAM in the *.data* section when initialized, followed by the *.bss* section, where the uninitialized variables are mapped to. This is a common binary representation format of the different memory sections. When a new global variable is initialized, all other variables located in subsequent RAM locations will be shifted and the references (by other functions) to them will have different target addresses for the two firmware images. Hence, a declaration of an initialized global variable will effectively shift all variables in the *.bss* section and the ones following within the *.data* section. Moreover, a code programmer has no control over the placement of the global variables in RAM, since this is determined by the compiler and not by the order they are declared in the source code [78].
- **Relative jumps (inside functions)**. Inside the source code, the jumps from one function to another are performed using an offset value that represents the distance between the jump instruction and the target address. For this reason, inserting new instructions or deleting others between a jump instruction and its target, can result in a different offset, which requires additional *ADD* instructions for the encoding.
- **Indirect addressing**. In RISC architectures, memory locations can only be accessed indirectly through the processor registers. For example, if an instruction calls a function, this function's address will be stored in a register, and then the control circuit will enforce a jump based on the value this specific register contains. Since, as stated above, the functions and global variables can be shifted in an update, registers' values need to be updated; thus, additional *ADD* operations are required.

Based on the above, it is important to preserve images' similarity, mitigating the effects of function and variables shifts, prior to calculating the common segments, in order to create small-sized delta scripts. Various techniques are proposed in the literature for keeping the target addresses unchanged. Some contributions presented later in this section transmit only the segments that have been modified in the new version and not the whole update image. In these schemes, when a node receives such a firmware segment, it either re-links the image locally or patches the image using the received information. Moreover, these updated segments can also be used as input to a differencing algorithm to further reduce the update overhead.

Most OTAP contributions avoid direct modifications of the compilers and linkers, as they are provided by hardware vendors, designed to deliver highly optimized code [55]. Usually, the files produced during code compilation and linking are used. For example, in [81], the MAP file along with the binary files produced before and after the static linking are used. A MAP file contains the relative offset and length of each object in the firmware and can be produced by

the linker, providing additional valuable information regarding memory usage [6]. Furthermore, the binary files produced during the static linking can provide information regarding code relocations, a useful input for schemes aiming to improve firmware similarity. Finally, many related works (e.g., [78, 79, 90]) target specific boards or platform families, although claiming platform independence. Nevertheless, platform independence cannot be achieved easily, as different microcontroller families use different re-location types and their definitions should be taken into account by the corresponding scheme. Platform-specific techniques have also been developed (e.g., [11, 41]). For example, the authors in [83] describe the effort for porting the Deluge protocol to support the Imote2 sensors, which was initially designed for the Tmote sky [1] and MicaZ [8] motes. In the next sections, we briefly describe techniques used to improve the similarity of firmware images. Moreover, in Table 2 some OTAP schemes are presented, along with the similarity-preserving technique and the differencing algorithm they have utilized, as well as some other characteristics.

3.1.1 Slop Regions. The use of the *slop regions* was first introduced in [60], and since then, they have been adopted by other OTAP schemes as a way to address function and variables shifts. A *slop region* is defined as the free memory space, located immediately after a function's code in the flash memory, where a function can grow or shrink without causing any other functions to relocate. If a function grows, part of its slop region will be utilized, thus without causing any other function shifts. If a function shrinks, its slop region will grow, occupying the removed part of the function, so other functions that follow will not be shifted. Besides the *.text* region that is mapped in the flash memory, *slop regions* have also been used between the *.data* and the *.bss* sections in RAM to avoid global variable shifts [78]. Finally, in order to implement this feature, the linker has to be modified with the risk to downgrade the produced code performance.

A more efficient use of the slop regions is presented by the authors of the Qdiff OTAP scheme [90]. To deal with function shifts, Qdiff does not create slop regions for each function during linking like other implementations do, but when a function is deleted or shrunk, the resulting available space becomes a *slop region*. Hence, *slop regions* can be found only immediately after functions' code or inside functions, as a result of removed instructions. On the other hand, if a function grows, Qdiff will try to find a *slop region* right after the function implementation, and if such a region exists, the function will expand there; otherwise, the new code will be moved at the end of the existing code. Moreover, as an update can create (or expand) some functions, while it can delete (or shrink) some others, Qdiff proactively creates empty slop regions that are later assigned to functions. A drawback of using *slop regions* is that excessive fragmentation of the memory space can occur, as some regions may contain code, while others remain idle. Apart from the inefficient use of the flash memory, fragmentation can increase the energy consumption because the control circuitry needs to activate a large number of memory regions. The authors in [55] show that the energy consumption can increase by up to 5% when memory is fragmented. Finally, extra care is needed when a function grows beyond its slop region and might need to relocate to a completely different memory region.

3.1.2 Position-Independent Code (PIC). Position-independent code is an option that can be set during code compilation, where code is compiled to execute normally, regardless of the absolute memory address it is stored in. All references and target addresses are related to the memory address of the calling instruction; thus, if shifts occur, the "relative" target addresses are not affected. Nevertheless, due to hardware limitations of the embedded devices, these relative (instruction) jumps can be performed only within certain offsets. For example, the Atmel AVR platform supports PIC but restricts the size of the program to 4 KB. The PIC technique is used by the SOS operating system [46] in order to avoid the effect of address shifts.

3.1.3 Indirection Tables. The *indirection tables* technique was first proposed in [78, 79] as a countermeasure against function shifts. During the firmware image linking, an indirection table is created and stored in a fixed location within the flash memory. In this table, there is one entry for each function called at least once, along with the memory address it is stored in. Any calls to the functions are replaced by suitable jumps to the corresponding entries of the table.

The advantage of this technique, as a similarity-preserving mechanism, is that when a function relocates, only its entry in the indirection table is affected (memory address updated), while the calling instructions are not affected. However, this technique is platform specific and linker modification is required. Moreover, because the function calls are performed indirectly, through the table, the runtime latency of the function call increases. Finally, the table size is proportional to the number of functions called, so in complex programs with many function calls, a large amount of flash memory is occupied (by the table).

3.1.4 Interrupt Service Routines Pinning. Interrupt service routines are software methods, invoked by hardware, to respond to specific interrupts (e.g., packets received by the network adapter). The memory addresses for these services are stored in an interrupt vector table, which in most embedded systems is placed at the beginning of the program memory. Whenever an interrupt occurs, the control goes to a pre-defined entry of the vector, and through it, the correct interrupt service routine is invoked. However, modifications in the firmware code can relocate these service routines, affecting the memory addresses contained in the interrupt vector table. The authors in [79] address this issue by mapping the interrupt service routines to fixed memory locations in the program flash.

3.1.5 Global Variables' Address Pinning. This technique was proposed in *Hermes* [78] as a way to ensure that the global variables appear in a specific order and hence are stored in the same address in each firmware version. The actual order of the global variables in the RAM memory is determined by the compiler type. This technique exploits the fact that members of a structure are placed in the same order in RAM, exactly as they are declared in this structure. The defined, as well as the undefined, global variables are detected and stored in two distinct structures, so if the update does not define additional global variables, it is ensured that the memory addresses of the current variables are not affected. Furthermore, Hermes utilizes a *slop region* between the *.data* and *.bss* sections to avoid address shifts of the undefined variables when the *.data* section shrinks or expands. In another related contribution [90], the authors follow a different approach to address data shifts. They modified the method for the *.data* and and *.bss* sections expansion in RAM. Instead of expanding toward the same direction, the two sections expand toward opposite directions. To implement this concept, the two sections are placed in a fixed address (initial address) with a large empty space between them (in RAM); thus, when a new initialized global variable is created, it is placed at the bottom of the *.data* section, whereas when an uninitialized one is created, it is placed at the beginning of the *.bss* section, so no data shifts occur.

3.1.6 Relocatable Code. When building a runnable program, such as a firmware image, various modules must be compiled and linked together to construct the final executable. These modules, referred to as *relocatables*, are initially assembled at pseudo-addresses, and when linked together, the linker resolves these address references to the correct values. However, once the final program is created (after linking), it should also be able to execute from different memory addresses, as multiple users may wish to run multiple instances of the same program; hence, *relocatable code* is a piece of software whose execution address can be dynamically moved around the available address space and loaded in multiple addresses. In addition, a relocation table is created, containing

all these memory references, and is used by the loader to resolve the references to the correct absolute addresses when the program executes.

In [29], the authors utilize the *relocatable code* technique to mitigate the effect of function and variable shifts. The key idea is to change all references to symbols (functions and global variables) to the same (predetermined) value and provide the needed metadata, so that the loader at the receiving side properly resolves the references before code execution. The pipeline the authors follow to create the firmware image consists of a three-stage process: First, the linker produces *relocatable code*, where the resulted relocation table contains an entry for each reference. Then, the target addresses of all reference instructions are changed to zero. Finally, the relocation table and the altered image are merged to form the final image that will be disseminated. In [31], the authors use the *relocatable code* technique more efficiently, minimizing the metadata transmission overhead. The authors observed that many instructions reference the same symbols; so instead of filling these references with zeros, they are filled with the corresponding symbol index, which is a reference to a symbol entry that contains the actual address of the symbol. Moreover, this scheme manages the symbols and ensures that their index will not be altered between different firmware versions. Finally, instead of using the 2-byte offset field to yield which memory location needs relocation, for compression reasons, a bitmap is used to indicate which 2-byte memory locations need relocation.

A key advantage of the *relocatable code* technique is that it handles more types of reference instructions in a more general way, and hence is able to perform better than other similarity-preserving techniques. A disadvantage, however, is that it requires a sophisticated loader that can resolve the relocated addresses before firmware executes, and the relocation table introduces additional transmission overhead. Finally, it must be mentioned that although *relocatable code* and indirection tables seem similar, they have some major differences. The *relocatable code* must be resolved during load time, while the indirection-table-based code operates in runtime, introducing an extra overhead. Furthermore, the metadata of the *relocatable code* are larger in terms of size compared to those of the indirection table-based code.

3.1.7 In-place Patching. The authors in [107] propose an OTAP strategy based on *in-place* code updating, using code patches to avoid system reboots and make use of the available memory more efficiently. The key idea is to transmit only the parts of the firmware that have been altered (patches), and the update module that executes on the receiving side copies them directly to the flash memory. A risk with this technique is that because the firmware image is updated in real time, the status of the firmware stored in memory can fall into an inconsistent state in case of failures (e.g., transmission errors). A countermeasure is to make all instructions that contain references to functions that are being updated halt till the update operation completes. Furthermore, when a new firmware update is released, multiple patches may be transmitted, each one replacing a specific part of the current firmware. Such a patch could also target the firmware code responsible for the updating; hence, code modifications must take place atomically, even when multiple patches are involved in the process. As mentioned before, the code segments being updated must be suspended to avoid the node falling into an inconsistent state. However, code halting till all patches are applied can cause a significant delay, similar to that caused when a node reboot is required. A countermeasure is to minimize the required memory writes (called the *atomic update set*) during the creation of an updated firmware. Two code patch strategies are proposed in [107]: (1) *in-place patching* and (2) *in-place patching with trampolines*. In *in-place patching*, all new and modified code parts are stored in the free space of the flash memory. The code insertion is performed by adding a jump instruction in the original code that has as target address the location where new code has been inserted. Respectively, at the end of the inserted segment, another jump instruction exists

that jumps back to the next instruction in the original image. Similarly, code deletion includes a jump from the address of the first deleted instruction to the instruction immediately following the last deleted one. Hence, the atomic update set contains all jumps added in the original code and not the modified parts that are transmitted, since the latter part of the code is placed at a safe place in the flash memory, where no conflicts can occur. This way, the size of the set is proportional to the number of the firmware parts required to be patched.

In *in-place patching with trampolines*, a code snippet, called a trampoline, is used for each patch. While the modified code is inserted in an unused memory location, like in the previous technique, the original code does not jump to these patches (at least not directly). However, a jump instruction is inserted in the original code to redirect the control to the corresponding trampoline. Initially, the trampolines return back to the instruction that follows the jump instruction in the original code. This is accomplished utilizing a centralized approach that instructs all trampolines to jump back to the original code, so no conflicts can occur and hence the atomic set is empty. Afterward, a *base* variable that all trampolines check to determine their return addresses is modified and instructs the trampolines to jump to the corresponding patch (new code inserted in an unused memory location). Using this global variable as a centralized approach and being able to apply all patches in a single shot results in minimal downtime, due to the small number of instructions in the update atomic set. Essentially, the atomic set consists of just the update of the *base* variable, the pointer that directs all trampolines to jump to the corresponding patch.

3.1.8 Dynamic Linking of Modified Firmware Sections. The use of a dynamic linker [32] at the node side was introduced as part of the first version of the Contiki OS [33], an operating system for constrained IoT devices. In this OTAP scheme, only the modified sections need to be transmitted, since upon reception, the dynamic linker will re-link the image and load it again, replacing the previous one in the program flash. However, a limitation of this technique is that it requires an operating system or a sophisticated linker that can resolve the addresses properly. Moreover, the modified section is usually accompanied by the new symbol and relocation tables that will be used during the dynamic linking, increasing transmission overhead.

3.1.9 Modules Extraction. This technique was presented by the incremental firmware update algorithm *MoRE* [81]. In order to create the firmware image, many object files are linked together. The MAP file format is used, which is a text file format that presents the relative offset and length of each object in a binary file. The proposed method extracts binary fragments called modules from a firmware image. By comparing the modules extracted from two sequential incremental updates, it becomes feasible to detect which modules are modified. This was accomplished using modified versions of the *R3diff* [31] and *RMTD* [50] image comparison algorithms for the delta calculations. In evaluation experiments, although *MoRe* in most cases resulted in slightly larger data transmissions compared to the (modified) *RMTD* and *R3diff* algorithms, it performs better in terms of the required time to complete. It must also be noted that in *MoRe*, there is no need to transmit extra information such as metadata and relocation-indirection tables, thus resulting in lower network overhead.

3.1.10 Replaceable Components. The concept of *replaceable components* was introduced in the *Elon* reprogramming scheme [30] as a way to reduce the data code to be transmitted during a firmware update (for the TinyOS operating system). *Elon* is based on the assumption that TinyOS kernel components (e.g., CTP, FTSP) are rarely updated, in contrast to the application components; hence, the programmer can define in the source code which components (code and data) can be updated in a future version. To do so, programmers have to use the appropriate annotations in the source code of the base firmware (e.g., *@replaceable*, *@system*).

In *Elon*, the *replaceable components* and system components are stored in different sections, *.vdata/.vbss* and *.vtext* for replaceable data and code, respectively. Initially, when the first (base) version of the firmware is created, the replaceable sections are stored in the flash memory and are not removed throughout the node's lifetime, as they serve as a *golden image* to be used for fail-safe operations. In subsequent firmware updates, the *replaceable components* that are received by a node are stored in the RAM to avoid further flash memory access. In order for the linker to determine the base address of these sections, a two-phase linking process is required. During the first phase, the firmware is compiled and the size of these sections can be determined, since they do not involve external libraries that may be invoked after linking (external libraries cannot be replaceable). Given this, the linker is able to find the starting addresses of all sections in both the RAM and flash memory. Moreover, Elon utilizes an indirection table that is called a *jump table*. This is crucial, because the *replaceable components* can be relocated and the indirection table helps to mitigate the effect of these shifts. Based on this technique, *Elon* is able to update firmware without the need for a node reboot. A kind of software reboot process takes place, where, initially, the replaceable sections and the new jump table are stored in RAM, and then the control jumps to the initial address of the firmware image. This way, no kernel data initialization is required, minimizing the required downtime. An advantage of *Elon*, over other OTAP schemes, is that it does not require sophisticated OS support (e.g., dynamic linking). Moreover, it does not produce additional metadata (e.g., indirection or relocation tables). However, a concern exists with regard to the size of RAM required to store modern and complicated firmware. Moreover, as the *replaceable components* are stored in RAM, they are not persistent and have to be re-transmitted in case of a hardware reset. Finally, in *Elon* it is assumed that the core system code and the libraries will not be updated in future firmware versions.

3.2 Differencing Algorithms

As described in the previous section, several techniques exist that preserve the similarity of the two firmware images, mitigating the effects of the function and variable shifts. When this process has been completed, it is desirable to minimize the required data that needs to be transmitted for the update of the IoT nodes. To make this feasible, the sending station (e.g., a firmware server) that initiates the firmware update uses a *differencing* algorithm in order to find the common segments between two firmware images (the firmware that the nodes currently run and the updated one). Differencing algorithms can be of two types; either block level or byte level, depending on the granularity level at which they are able to detect matching segments.

Block-level algorithms (e.g., [54, 100]) split the firmware images into fixed-size blocks, aiming to detect non-common segments between the two images; hence, their accuracy is highly affected by the block size. On the other hand, the byte-level algorithms (e.g., [31, 50]) are able to find non-common segments between two firmware versions using blocks of variable lengths and can utilize more fine-grained approaches in order to achieve better accuracy, for example, dynamic programming [50]. Regarding algorithms' performance, the block-level ones can detect a limited number of non-common segments, since they are not able to detect those with size smaller than the size of a block. However, these algorithms typically have smaller time and memory footprint. Byte-level algorithms, on the other hand, can detect more non-common segments but typically require more time to complete. With respect to the limitations of the IoT-constrained nodes, it is evident that in order to utilize the available resources more efficiently, the required amount of transmitted data during a firmware update has to be minimized. This is achieved through a process referred to as *delta script generation*, which exploits two principles: (1) the nodes to be updated have already stored a previous firmware version in their flash memory, and (2) the updates mostly introduce small modifications to the firmware binary code. The key idea of the delta scripts is to transmit only

Table 1. Summary of Differencing Algorithms Commonly Used for OTAP Schemes (N Is the Combined Length of the Two Firmware Images in Bytes)

Algorithm	Type	Time Complexity	Space Complexity
FBC [54]	block-level	$O(n)$	$O(n)$
Rsync [100]	block-level	$O(n^2)$	$O(n)$
RMTD [50]	byte-level	$O(n^3)$	$O(n^2)$
Hirschberg's trick [72]	byte-level	$O(n^2)$	$O(n)$
R3diff [31]	byte-level	$O(n^3)$	$O(n)$
DASA [73]	byte-level	$O(nlogn)$	$O(n)$
DG [57]	byte-level	$O(n^2)$	$O(n)$

the parts of the firmware that have been altered, followed by instructions (for the node) regarding the local firmware reconstruction. Hence, the common and non-common segments detected by the differencing algorithm, along with some special-purpose instructions, are encoded into commands of a *delta script* that is transmitted to the receiving nodes. Based on this script and the current firmware code, each node is able to re-construct the new firmware version locally, executing the commands in the script. It is evident that the delta script creator should be aware of (1) the current firmware image a node currently executes and (2) how the current image is mapped in the internal memory because the encoding highly depends on these two.

There has been an effort to create universal delta script formats, e.g., VCDIFF [59], as well as some other custom ones [29, 31] that support additional instructions, in order to achieve a more efficient data encoding. Despite their differences, all proposed formats feature two common core operations with relatively standardized syntax: *COPY* and *ADD*. The *COPY* [79] instruction is used to encode the segments of the new image that have been matched with others of the current version. The update module at the receiving side, upon interpreting the *COPY* instruction, copies a segment with a pre-defined length from the current firmware image. Moreover, the starting address of the sequence must be provided. Once the update module has determined the length and the starting address of the copied segment, it copies and appends it to the new firmware image (that is reconstructed locally). The *ADD* [79] instruction is used to encode the segments of the new image that do not match with other segments of the current version, and hence, they have to be fully transmitted. When the *ADD* instruction is interpreted by the update module at the receiving side, the associated received sequence will be used (appended) to reconstruct the firmware image. There are a few more other instructions such as *PAD* [56], *RUN* [59], and *REPEAT* [79].

Some of the most popular differencing algorithms are shown in Table 1 and analyzed in the following sections.

3.2.1 Fixed Block Comparison (FBC). *FBC* [54] is the simplest method for comparing two firmware images, aiming to minimize the required data transmission for an update. This algorithm splits the two images into blocks and then compares each corresponding block. For each matching block pair, a *COPY* instruction is inserted into the produced delta script, while the non-matching ones are transmitted along with the delta script. In order to encode the latter blocks, an *ADD* instruction needs to be inserted in the delta script. The main benefit of this technique is the low time and space overhead, as well as the ease of implementation. Moreover, it works well for small firmware changes, since only the altered blocks are transmitted. Nevertheless, it operates at block-level granularity and is not able to detect a high number of common pairs, especially when the update includes excessive modifications.

3.2.2 Rsync. *Rsync* [100] is an algorithm used by many incremental reprogramming schemes (e.g., in [53, 79]), in order to compute the common segments of two firmware images, initially developed for binary files exchange over low-bandwidth channels. This is a block-level differencing algorithm that splits the firmware images into fixed-size blocks and then uses a sliding window with a size equal to the block size to scan the two firmware images for detecting matching segments. Initially, a {Checksum, MD4} pair is calculated for each block of the current firmware image, and then the window traverses the new image, the {Checksum, MD4} pair of each window is calculated, and look-ups with the pairs of the current image are performed to detect potential matches. Like typical sliding window protocols, when a match is found, the window moves forward one block; otherwise, it moves one byte, labeling this block as unmatched. All unmatched blocks are accumulated for transmission either when a next block is matched or the current window reaches the end of the new image. Although *Rsync* is able to find subsequences with a higher accuracy compared to *FBC*, it still faces similar drawbacks, since its granularity depends on the window size used, thus being not able to detect common segments with a size smaller than that of the window used.

3.2.3 Reprogramming with Minimal Transferred Data (RMTD). *RMTD* [50] is a byte-level algorithm that aims to find the optimum combination of common sequences between two images, in order to minimize the number of transmitted bytes. Similarly to other differencing algorithms, *RMTD* aims to detect the common segments between the two firmware versions. Nevertheless, a novelty of this algorithm is that it utilizes the partially reconstructed firmware image to detect matching segments. As the instructions in the delta script are executed at the receiving side sequentially, the new firmware is gradually rebuilt. Hence, for each segment of the new firmware, *RMTD* checks if it can also find matching segments in this partially rebuilt image. *RMTD* uses a 2D matrix to record the pairs of the common bytes found for the two firmware images, with the comparisons performed in both forward and backward order to achieve higher accuracy. The result of this operation consists of two lists that contain the matching segments of the two images, as well as the matching segments between the partially reconstructed new image and the rest of the (new) image, respectively. Once these two lists are prepared, the algorithm finds the optimal combination of the *COPY* and DOWNLOAD instructions to encode the common segments. Regarding the segments that can be encoded using *COPY* instructions (found in any of the two lists described above), many such common sequences may correspond to the same memory addresses. When the respected *COPY* instructions found in the delta script are executed, they will result in multiple writes of the same data to the same memory addresses; hence, these redundant writes lead to energy waste and also affect flash memory life span. To mitigate this issue, *RMTD* finds the optimal combination of *COPY* instructions on the detected common sequences, using a dynamic programming approach. Finally, it must be noted that the algorithm's complexity depends on the size of the images, which makes it unsuitable for increasingly complex programs, as the time required to complete is substantially high. Moreover, in experiments conducted by other researchers (e.g., [73]), it was shown that *RMTD* crashes when the code size becomes too large ($\tilde{4}$2 Kb), due to lack of memory.

3.2.4 Hirschberg's Trick. *Hirschberg's trick* [48] is a method for computing the longest common sequences between two strings while saving space utilizing a dynamic programming approach. A **longest common subsequence (*LCS*)** of two strings $X{=}x_1x_2x_3\ldots x_m$ and $Y{=}y_1y_2y_3\ldots y_n$ is a subsequence of both X and Y, whose length is the maximum possible. Let all prefixes of the two strings X and Y be $\{X_1, X_2, X_3, \ldots X_m\}$ and $\{Y_1, Y_2, Y_3, \ldots Y_n\}$, respectively, where X_i and Y_i represent the prefixes that contain the first i bytes of the corresponding string. Moreover, between two prefixes X_i and Y_j, there may be multiple longest common prefixes, but all of them will have equal length; hence, if we denote the length of the *LCS* of these prefixes as $C(i, j)$, its dynamic

formulation is as follows:

$$C(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C(i-1, j-1) + 1 & \text{if } x_i = y_j \\ max(C(i-1,j), C(i,j-1)) & \text{if } x_i \neq y_j. \end{cases}$$

Using the above formulation, the common subsequences between the prefixes of two images can be found in order to compute the delta script. Based on *Hirschberg's trick*, the authors in [72] proposed a differencing algorithm to detect common firmware subsequences. Hirschberg also presented a modified version of this algorithm, which follows a divide-and-conquer approach and is able to compute the *LCS* of two strings in O(min(m,n)).

3.2.5 R3diff. R3diff [31] is a byte-level comparison algorithm that complies with the overall design of the R3 OTAP scheme. Initially, the algorithm computes the hash values for every three continuous bytes of the current image. Three bytes were chosen as the lowest level of granularity because copying smaller byte segments (e.g., two bytes long) is not more beneficial than adding them, due to the overhead of the *COPY* instructions that comes with additional parameters in the delta script. In order to compute the optimal delta for transmission, the authors use the opt_i annotation that represents the minimum delta script size that needs to be transmitted, in order for the first i bytes of the update image to be reconstructed by the receiving node. Moreover, a *findK* method is used for each such prefix (first i bytes) that returns the smallest index k, so that the subsequence $[k,i-1]$ is a common segment (found in both the new and current images). Furthermore, two additional notations are used by the algorithm, opt_i^A and opt_i^C, that represent the minimum number of bytes that need to be transmitted in the delta script in order to reconstruct i bytes, having as last instruction an *ADD* or a *COPY* one, respectively. Hence, in order to reconstruct the first i bytes, $opt_i = min(opt_i^A, opt_i^C)$, omitting the overhead of *COPY* instructions, as encoded in the formulations below. To compute the optimal opt_i, the algorithm follows a recursive approach, starting with $opt_0 = 0$, iterating over all possible prefixes, till the final one that corresponds to the new image. For each prefix, the algorithm computes opt_i^A and opt_i^C and also runs the *findK* method to check if the prefix can be encoded in a *COPY* instruction using a common segment. If no such common segment is found, opt_i^C is set to a large integer, so that it is not selected over opt_i^A. The formulations of opt_i^A and opt_i^C are as follows:

$$opt_i^A = min\left(opt_{i-1}^A + 1, opt_{i-1}^C + \alpha + 1\right)$$

$$opt_i^C = \begin{cases} LARGE\_INTEGER & \text{if } k > i-1 \\ opt_k + \beta & otherwise, \end{cases}$$

where α is the additional overhead imposed by each *COPY* instruction and β the overhead of an *ADD* instruction.

3.2.6 An Efficient Differencing Algorithm Based on Suffix Array (DASA). DASA [73] is a differencing algorithm that focuses on minimizing the space and time complexity for computing the optimal delta script. In order to accomplish this, it utilizes an efficient data structure, called ***suffix array (SA)*** [25]. *SAs* can be used for data processing, as well as for data compression operations. Initially, the algorithm combines the two firmware images in a **$-#**-padded extension format, using them in reverse order. For example, two strings S_1 = "cdn" and S_2 = "ngtv" will result in T = "ndc#vtgn$". Once the extension is created, the doubling algorithm [25] is used to compute the *SA* by sorting all possible suffixes in ascending order and for each suffix stores its starting index in T. After the computation of *SA* is completed, it is used by *DASA* as input to create the height

array in time complexity of $O(nlogn)$. The height array contains the length of the longest common prefix of each suffix with the next one in sorted order. Having computed the height array, one is able to get the LCP of any two suffixes in linear time. Next, the algorithm computes the optimal delta script, annotating as opt_i the optimal delta script size to reconstruct the first i bytes of the new image ($opt_0 = 0$). For each prefix of the new image, *DASA* uses the *findK* method (discussed in Section 3.2.5) to find the largest common segment that can be used to encode the prefix using *COPY* instructions. Moreover, *DASA* utilizes the formulation of opt_i^A and opt_i^C, similarly to *R3diff*, in order to find the opt_i. Therefore, *DASA* iterates over all possible prefixes to find the optimal combination of *ADD* and *COPY* instructions. The experimental evaluation shows that *DASA* outperforms *Rsync* in terms of the delta script size, which is expected, as *Rsync* is a block-level algorithm. Despite several improvements introduced by the *Rsync* developers, *RMTD* and *DASA* still have superior performance. Furthermore, *DASA* has better performance than *RMTD* both in time and space domains and this stands true especially when the new image is relatively large in size.

3.2.7 Delta Generator (DG). Many differencing algorithms assume abundance of spare memory [50] at the node side for firmware reconstruction, something not always true due to the constrained nature of several IoT node types. In [57], the authors propose a differencing algorithm, known as *DG*, for nodes that lack external memory. The algorithm, by exploiting the fact that two sequential firmware versions usually share many common parts, places the two images side by side and executes an *XOR* operation between the corresponding bytes, aiming to reveal the sequences of the non-matching bytes. The matching sequences can easily be encoded using *COPY* instructions in the delta script, and the non-matching ones can be further broken down into matching and non-matching subsequences to achieve delta script size minimisation. To make this feasible, each non-matching sequence is checked against the current firmware to find matching subsequences. The common subsequences found are reconstructed using *COPY* instructions. This algorithm has $O(n+m)$ space complexity and $O(nm)$ time complexity, where n is the size of the current firmware image and m the size of the non-matching segments. A comparison of *R3diff* and *DG* was conducted in [65] using various image sizes and code shifts. The authors inferred that *DG* generates significantly smaller delta scripts than *D3diff* for small-sized images, but this does not stand true as more data and code are shifted. Moreover, the authors found that the number of *ADD* instructions in the delta script gets smaller as code shifts increase. The authors conclude that *DG* is not able to provide optimization for a high number of small changes; instead, it generates a number of *ADD* instructions that encode regions with a few bytes for each non-matching segment. When the code shifts increase, these non-matching segments expand together and are merged under one common *ADD* instruction. This results in a larger delta script with fewer *ADD* instructions.

3.3 Dissemination Protocols

Traditional data dissemination protocols (e.g., [61, 75, 93]), initially designed for WSNs, are not suitable for firmware dissemination in IoT networks for a number of reasons. First, the size of the update image is typically larger (in the order of kilobytes) than that of the commonly transmitted data, while the corresponding protocols have been specifically designed to propagate small size packets with a low packet rate. Furthermore, during the update process, the network nodes usually store the update image and then act as sources, which is not a typical behavior in data dissemination protocols. Finally, while the flow of data transmissions in a WSN is bidirectional, including operation information from sensors and commands toward actuators, the flow of the update image is one-way, from the base station to the network nodes. Hence, protocols designed for disseminating firmware updates in WSNs focus on the efficient distribution of the new firmware code from a

Table 2. Summary of OTAP Schemes

OTAP Scheme	OS/Platform	Update Type	Firmware Similarity	Differencing Algorithm	Live Update	Dissemination Protocol
Elon [30]	TinyOS, TelosB	Incremental (replaceable components)	Replaceable components	-	Yes	Deluge
R2 [29]	TinyOS, TelosB	Incremental (delta script)	R2sim (Relocatable code)	RMTD	No	Stream
R3 [31]	TinyOS, TelosB	Incremental (delta script)	R3sim (Relocatable code)	R3diff (based on DASA)	No	Stream
MoRE [81]	NanoQplus OS, Mango-Etoi board	Incremental (modified modules)	-	RMTD, R3diff	No	-
Zephyr [79]	TinyOS, Mica2	Incremental (delta script)	Indirection tables	Rsync	No	Stream
Hermes [78]	TinyOS, Mica2	Incremental (delta script)	Indirection tables, global variables pinning	Rsync	No	Stream
In-place patching [107]	TinyOS, MSP430FR5739	Incremental (Patches)	In-place patching (with or without trampolines)	Rsync or Zephyr	Yes	-
Qdiff [90]	TinyOS, IRIS mote	Incremental (delta script)	Slop regions, RAM layout modification	Google-guava API, Google diff-match-patch	No	-

central firmware repository server to the nodes. These protocols have to cope with the unreliable nature of the wireless communication medium, employing suitable mechanisms for the provision of a reliable firmware update process. Regarding this, the nodes should be able to provide a form of feedback, indicating the correct reception of update (network) packets, as well as to request the retransmission of lost ones. The simplest method to disseminate a new firmware image is by flooding, where the firmware server broadcasts the new firmware code to its connected IoT nodes, and the latter further broadcast it to their neighbors in an epidemic fashion. However, redundant transmissions during flooding should be minimized, as they can easily deplete nodes' battery and can cause the *broadcast storm problem* [101], where overlapping radio signals result in an increased contention and packet collisions.

In order to minimize the number of messages required during a firmware update, several protocols are proposed that take into consideration the underlying network topology. For example, in [12], a broadcast protocol is presented that provides reliable data propagation within the network, by splitting the nodes into several clusters. As wireless networks suffer from problems like collisions and the hidden terminal problem, in order to minimize the required transmissions, most dissemination protocols instruct nodes to aggregate data prior to transmission to their neighbors. Moreover, they typically use a three-way handshake pattern to establish a communication channel between the firmware repository server and the nodes. Initially, the available sources advertise the available firmware version by broadcasting a message to their neighbors. Since some nodes may receive multiple advertisements of this type, they select a specific node as the firmware source

based on some heuristics and then broadcast a request packet directed toward this selected source. Once the request packet is received (by the source), the actual data transmission begins. When a node has partially or fully received an update image, it can broadcast an advertisement message, indicating that it can now act as a firmware update source. The dissemination protocol can also be based on a subscription approach (e.g., [94]), where the nodes of the network subscribe to sources in order to receive firmware updates. This can create an additional overhead for the sources, which are also nodes of the network, as they have to track all subscriptions. Finally, most update dissemination protocols require a form of feedback from the nodes to the sources in order to validate packet correct reception. This can be accomplished by either ACKs or selective NACKs (negative ACKs). In the first approach, the node, upon successfully receiving a packet, transmits a short-length ACK to the sender. If the source has not received an ACK from a node within a predefined time interval, it re-transmits the packet. Typically, the source tries to transmit lost packets several times and quits after a number of failed attempts. As every packet has to be ACKed separately, the network implosion problem is possible in case of a large number of missing packets. To deal with this, several contributions use the NACK option, where a NACK is sent from a node to the source only if a packet has not been successfully received, effectively reducing the number of control packets that need to be broadcasted during the firmware update process.

Several update dissemination protocols are proposed with the characteristics stated above. Some of them also use *pipelining* (e.g., [24, 28, 44]), a method that allows the parallel transfer of data within the network, thus achieving better performance in terms of the time required to transmit a whole firmware image. Initially, the image is split into segments and dissemination is performed segment by segment. When a node completely receives a segment, it can become a source and can further disseminate it to its neighbors. In order to validate or compare the performance of the dissemination protocols, authors typically utilize simulation frameworks (e.g., TOSSIM [66], EmStar [36]) or even use real-world sensor deployments as empirical testbeds (e.g., [51, 62, 67, 80]). The latter can provide more accurate results, as the experiments are usually conducted in various realistic indoor and outdoor environments. However, in order to stimulate congestion and collisions, a large number of networking devices is required, drastically increasing validation cost. To this end, the authors use simulation frameworks, where they can simulate network topologies along with links' quality, thus being able to increase the number of the nodes and observe the scalability and behavior of the proposed protocol. Some of the most commonly used protocols in OTAP schemes are presented in the following sections (summarized in Table 3).

3.3.1 Trickle. *Trickle* [67] is an update dissemination algorithm built for the TinyOS and the Mica-2 motes [47], following a "polite gossip" approach to propagate the new firmware code throughout the nodes of a network. In *Trickle*, each node periodically broadcasts an announcement that contains the current firmware version this node executes, informing others about a potential update. Firmware update propagation takes place if a node overhears that another node can provide a newer firmware version or if a node receives a broadcast by another one indicating that it executes an older version. This "polite gossip" approach makes *Trickle* robust and scalable, able to operate in various networking environments (sparse, dense network topologies). Moreover, when a node overhears that a neighbor broadcasts metadata for an outdated firmware version, it broadcasts the code of the newer version, initiating the update of the outdated node. Each node breaks time into fixed-size intervals, and at a random point within an interval, it broadcasts announcements (metadata) that inform other nodes about the firmware version it can provide. However, if a node has overheard several other nodes broadcasting the same metadata, it stays quiet, since acting as a source can cause redundant transmissions, wasting network resources. When a node overhears such an announcement, there are two possible cases: (1) the node executes a newer

Table 3. Dissemination Protocols for OTAP

Protocol	OS/Platform	Pipelining	Dissemination Hops	Encoding	Feedback /Reliability Strategy	Experimental Validation
Trickle [67]	TinyOS, Mica-2	No	Multi-hop	Full image	NACK-based	Simulated (TOSSIM)
Deluge [24]	TinyOS, Mica-2	Yes (pages)	Multi-hop	Full image (can also support incremental updates)	NACK-based	Simulated (TOSSIM)
Rateless & ACKLess Deluge [44]	TinyOS, Tmote Sky	Yes (pages)	Multi-hop	Full image (can also support incremental updates)	FEC	Simulated (TOSSIM) and test-bed
MOAP [94]	TinyOS, Mica-2	No	Multi-hop	Full image (can also support incremental updates)	NACK-based	Simulated (EmStar) and test-bed
MNP [62]	TinyOS, Mica-2 and XSM	Pipelined (Segments) and non-pipelined	Multi-hop	Full image (can also support incremental updates)	NACK-based	Simulated (TOSSIM) and test-bed
XNP [98]	TinyOS, Mica-2	No	Single-hop	Full image	Queries originated by the base station	Test-bed (Mica-2)
CORD [51]	TinyOS	No	Core-based	Full image	No feedback	Test-bed
ACDP [28]	TinyOS, TelosB	Yes (pages)	Multi-hop	Full image	NACK-based (Unicasted)	Test-bed
Stream [80]	TinyOS, Mica2	Yes (pages)	Multi-hop	Full image (can also support incremental updates)	ACK-based	Simulated (TOSSIM) and test-bed

firmware version than the one broadcasted, and (2) a node executes an older version than the one broadcasted. In the first case, the node will respond by broadcasting its (newer) code, while in the second case, the node will trigger a firmware update process by broadcasting its (old) firmware version, so the node that initiated the "gossip," as soon as it receives this message, will broadcast its (new) code. Moreover, instead of using a fixed transmission rate per node, *Trickle* dynamically regulates it by considering network density in order to achieve the desirable communication rate, permitting a specific number of transmissions in each interval and for each node. Network nodes must in advance synchronize the time intervals for their broadcasts; otherwise, *Trickle* could suffer from the short-listen problem; some nodes of the network exchange metadata immediately after the starting of their (time) interval period, listening for a short period of time, before any other node is able to broadcast its metadata. This can result in redundant transmissions because a node may never hear some other broadcasts and hence will not suppress its transmissions. *Trickle* overcomes this challenge by requiring nodes to fall into a listen-only state under which they cannot broadcast any metadata. The other half of the time interval is then available for each of the nodes to broadcast its metadata. Finally, *Trickle* was one of the first developed update dissemination protocols, and since then, it has been adopted as the basis for many dissemination protocols (e.g., Deluge [24, 62]), with many contributions aiming to further optimize it (e.g., [26, 42]).

3.3.2 Deluge. Deluge [24] was built for the Mote-2 devices as the default network reprogramming protocol for TinyOS. *Deluge* employs a negotiation mechanism based on *Trickle*, but with an increased performance, as it provides pipelined firmware dissemination; hence, when a node has received a chunk (page) of the update image, it can also act as a source for it, serving its neighboring nodes. A firmware image prior to its dissemination is split into pages, and each such page is further split into fixed-size packets that fit to the maximum packet size of the TinyOS network stack. Using *Deluge*, nodes periodically advertise the pages of a firmware version they can provide through broadcast packets, while other nodes send requests for pages they are missing and are willing to receive. The available pages a node holds for a specific firmware version are represented by a bit vector that is carried by the advertisement packets. *Deluge* enforces a sequential transmission of the pages, so for a node to request a missing page, it is required to have successfully received all previous ones. When a node receives an advertisement packet and infers that there is a new firmware version available, it first finds the lowest numbered page that it needs to receive. In most cases, where the firmware image binary has been completely changed, this page will be the first one. Once this page is determined, the node waits for a predefined time interval to receive further advertisements transmitted by neighboring nodes and to decide which of them can provide the specific page. When this period is up, the node heuristically selects one of the available source nodes and transmits a request packet that indicates the page and the packets within the page that it wants to receive.

The selection of the most appropriate firmware source node for a specific page is a challenging task, as the total energy consumption and network bandwidth utilization should be minimized. Heuristics used by *Deluge* aiming to address these challenges are as follows: a node requests data from the source node that transmitted the most recent advertisement packet; a node requests data from the nearest source node; a node requests data from the node farthest from the source, in order to inflate the overall spatial multiplexing; a node requests data from the node nearest to the source node. The latter heuristic was used only for comparison to others, since it has many disadvantages. For example, it neither tries to promote high-quality links nor tries to improve the spatial multiplexing for a better data propagation performance. Regarding *Deluge's* feedback mechanism, this is not explicitly based on ACKs or NACKs, because such packets are not transmitted, but the protocol itself is described as NACK based, because, by default, nodes re-request packets either not received or corrupted. Additionally, in order to decrease energy consumption and excessive bandwidth utilization, *Deluge* uses a self-organized suppression mechanism that allows each node to deactivate itself, based on packets it overhears; for example, it does not transmit a page request if, in the meantime, it overhears the same request sent by a different node. The *Deluge* disadvantage is that it requires nodes' radio to be always turned on, resulting in an increased energy consumption. Moreover, the protocol does not support fault detection or recovery mechanisms. The authors proposed an optimization based on **forward error correction (FEC)** using the *digital fountain approach* [21], a method for efficient transmission of bulk data by heterogeneous nodes.

3.3.3 Rateless and ACKLess Deluge. Rateless Deluge [44] is a firmware update dissemination protocol based on *Deluge* but introducing several modifications to its propagation mechanism, aiming to enable rateless transfer of the firmware image and to reduce the overhead due to lost packets' retransmissions. Using a *rateless coding* approach, the authors manage to minimize the amount of the control messages required. Additionally, the nodes do not need to explicitly specify which packets need to be retransmitted, as they only need to specify the number of packets successfully received. In order to construct rateless codes, the authors utilized the theory of ***random linear codes* (RLCs)** [49].

Rateless Deluge follows the same approach, splitting the firmware image into pages that are further divided into packets, and then encoding each packet using RLC. Respectively, a receiving node stores the encoded packets in its memory, and once the number of received packets has reached the page size, it proceeds to decoding using the ***Gaussian elimination process* (GEP)**. If decoding is successful, the linear equation system is solved and the specific page is stored in the flash memory of the node, while it is requesting the next page. However, if the received packets are linearly dependent, the process of *GEP* fails. In this case, the node discards the linearly dependent packets and waits for some time to receive more packets, and then starts *GEP* again. In contrast to *Deluge*, if some packets are lost, the receiving node informs the firmware source about the lost packets, without specifically identifying them, thus reducing the required feedback overhead. To further improve performance, *Rateless Deluge* exploits the fact that the pages are requested in an incremental fashion and pre-codes the next page.

ACKLess Deluge adopts the changes introduced by *Rateless Deluge* (on top of *Deluge*), but its main goal is to further reduce the need for retransmissions by employing an FEC algorithm, which operates at the packet level and appends extra encoded information to avoid additional control messages and retransmissions. The amount of redundant information is proportional to the calculated loss probability of each distinct receiving node.

3.3.4 Multihop Over-the-air Programming (MOAP). *MOAP* [94] is another multi-hop OTAP dissemination protocol, specifically designed for the Mica-2 motes, executing TinyOS. The goal of *MOAP* is to minimize RAM usage and energy consumption during a firmware update. This protocol does not offer pipelining, since for a node to become a firmware source, it has to receive the whole image in advance. The firmware update image is built using the standard TinyOS tools and then is split into a number of segments, with each segment transmitted using a single network packet. The *MOAP* dissemination mechanism is called *Ripple,* and unlike other dissemination algorithms, it avoids network flooding by selective firmware forwarding to other nodes, also utilizing a sliding window protocol for the identification of lost packets. *MOAP* uses a publish-subscribe mechanism for firmware dissemination in a neighbor-by-neighbor node fashion. A node that has received the full firmware and can become a (firmware) source advertises information regarding the type and version of the firmware image it already holds through a *PUBLISH* message that is broadcasted periodically. Interested nodes transmit *SUBSCRIBE* messages in order to subscribe for a firmware update. As soon as a source node receives such a message, it waits for some amount of time to receive any further subscriptions (in order to suppress duplicate transmissions) and then provides the new firmware image to the subscribed nodes, which can further become source nodes. These nodes wait again for some amount of time to receive any subscriptions, and if not, they reboot and the firmware update takes place. Eventually, all nodes are programmed with the new firmware version.

3.3.5 Multi-hop Network Reprogramming (MNP). *MNP* [62] is a multi-hop reprogramming protocol, designed for the Mica-2 and XSM [85] nodes running TinyOS. This protocol aims to reduce network collisions occurring during firmware dissemination by proposing a source node selection algorithm, which guarantees that at any given time, and for each neighborhood, at most one node can act as a firmware source. Additionally, *MNP* improves propagation performance by supporting pipelined dissemination, splitting the firmware image into fixed-size chunks and permitting nodes to act as sources for the pages they have already received. Each source node has a unique **identifier (ID)** and maintains a *ReqCtr* value that indicates the number of its receiving nodes (the nodes that have transmitted a request packet to this specific source node at least once). This value is incremented by one each time the source node receives a new request. At random intervals, each node broadcasts an announcement that contains its ID and *ReqCtr*, as well as the firmware version

ID. When a neighbor receives this announcement, if it is interested in this new code, it broadcasts a request that contains the *ReqCtr* and the ID of the corresponding source node. Using this technique, nodes unable to receive the initial source node announcement due to network problems (e.g., hidden terminal) can become aware of other nearby source nodes. Receiving nodes, upon the reception of broadcast announcements, send replies back to the corresponding source nodes that increase (by one for every reply) their *ReqCtr* values. The node with the highest *ReqCtr* value is selected as a source node that can later disseminate the firmware to the interested receiving nodes. The source node, after the completion of a firmware update, enters into a sleep state for some amount of time to enable a uniform load distribution, as other nodes can now have the chance to become source nodes. As soon as a new source node is selected, it broadcasts a *StartDownload* message to inform potentially interested receiving nodes that it has an available firmware image to provide.

3.3.6 XNP. XNP [98] is the earliest network reprogramming protocol hosted within the TinyOS operating system for the support of the Mica-2 motes. *XNP* transmits the whole firmware image (no support for incremental programming) and is able to disseminate it from a firmware server to only its single-hop nodes. Initially, the server splits the firmware binary code into multiple packets that are then broadcasted one by one. Single-hop nodes able to receive these packets store the carried information in their external memory. Nodes request any lost packets until the entire binary code is correctly received. To accomplish this, *XNP* at the firmware source node, checks the successful delivery of each packet by querying the receiving node. When the image is completely received, *XNP* uses the bootloader to copy the code to the flash memory and restarts the node [102]. A major drawback of *XNP* is that it does not support OTAP in multi-hop networks. Moreover, each time an update is required, the whole firmware image needs to be transmitted, as there is no delta mechanism supported. *XNP* also occupies a significant portion of the program memory in the (constrained) nodes. Finally, during the download of the update image, the application is halted and the *XNP* module that is wired into the firmware image executes. This introduces an extra overhead in terms of time, which is proportional to the network latency.

3.3.7 COre Based Reliable Dissemination (CORD). CORD [51] is a reliable bulk data dissemination protocol that mainly targets energy consumption minimization during the firmware update process. To achieve this, it follows a different approach compared to the other dissemination protocols presented in this article, which, due to their epidemic nature, propagate the update in a neighborhood-by-neighborhood fashion, employing also a three-way handshake pattern (advertise-request-data), resulting in a vast amount of control data. On the contrary, *CORD* follows a core-based two-phase approach, similar to the one used in Sprinkler [75] and Garuda [82]. The authors claim that it is possible to identify *reliable links* that have a constant low packet loss rate. Transmitting data over these links can result in less corrupted/lost packets and subsequently fewer control messages. During the first phase of *CORD*, a subset of nodes that are interconnected though reliable links is identified, forming an *approximate minimum dominating set* [43]. These nodes are selected as the *core nodes* using Cheng's *single leader algorithm* [23]. Subsequently, each network node can be either a *core node* or an immediate neighbor of such a node. Once the *core nodes* are selected, the update image is propagated from a firmware source to the *core nodes* through reliable multi-hop forwarding. The firmware image is initially split into pages and pipelining is used for increased performance. Once all *core nodes* receive the update image, *CORD*'s second phase begins, where the image is disseminated to the rest of the nodes. Furthermore, *CORD* uses a sleep scheduling schema to instruct nodes in the network that do not receive or transmit any data to turn off their radios in order to reduce energy consumption.

3.3.8 Adaptive Code Dissemination Protocol (ACDP). *ACDP* [28] has been developed for TelosB nodes executing TinyOS and aims to minimize the number of packets that need to be transmitted during the update image dissemination, providing reliability, low energy consumption, balanced traffic in the network, and rapid propagation. Similarly to *Rateless Deluge*, *ACDP* employs RLCs for efficient and robust data transmission. Prior to dissemination, the update image is split into several pages, and each page is further divided into a fixed number of packets. This new code image will be distributed by a single node, whereas the intermediate nodes will forward the pages they have received to their neighbors, incrementally, to ensure that the update will finally reach all network nodes. Additionally, at any given time, only one page is loaded into the RAM of a source node for transmission. When the node finishes with the transmission of a single page, the allocated RAM is released and it enters a sleep state, so that other nodes can transmit previous pages. In order to encode the packets of a page, *ACDP* is based on RLC, producing a linear combination of M packets of that given page. The receiving node uses the Gaussian elimination method to find the original page, which is a relatively efficient task, as described in Section 3.3.3. Hence, when a node has received enough packets, it attempts running the Gaussian elimination method, aiming to decode the initial packets of the corresponding page. If decoding is successful, the receiver chooses a sliding window of size N, depending on the number of its neighbors, and encodes N packets, computing their linear combination. This way, the node will send out M/N encoded packets into the network, achieving better load balancing within the network. Finally, nodes broadcast NACKs when a request has not been served within a pre-defined amount of time.

3.3.9 Stream. *Stream* [80] is an update dissemination protocol that aims to minimize the data transmission overhead by pre-installing the module responsible for the firmware update in nodes' flash memory as a distinct image. Hence, at any given time, each node stores two images: (1) firmware image and (2) reprogramming protocol image. The application image contains the actual firmware image and a limited reprogramming code (Stream-AS), whereas the reprogramming protocol image contains the code for the reprogramming/dissemination protocol (Stream-RS) and its purpose is to enable the firmware code update that is pre-installed in all network nodes. When a new update is released, all nodes are instructed to reboot to the stream-RS to permit the dissemination of the new firmware. To achieve this, a command is injected in the network from the base station that instructs the nodes to switch to the Stream-RS image. Once a node receives this command, it broadcasts this further to its neighbors and then reboots using the functionality provided by Stream-AS, which is included in the image that it is currently running. When all nodes receive the reboot command, they run the Stream-RS image, and the dissemination of the new update, augmented by the corresponding Stream-AS, is permitted. Furthermore, Stream-RS follows a three-way-handshake approach for the actual dissemination of the firmware that is based on the *Deluge* protocol. Each node contains a list of its neighbors that have transmitted firmware requests. When all these neighbors have successfully received all pages requested, the node is able to reboot from the application image, which now contains the updated firmware. After a certain period of time has elapsed, all nodes switch back to the application image, and thus all are updated with the new image.

4 OVER-THE-AIR PROGRAMMING SECURITY

Although IoT networks are often used in critical infrastructures, they are also known for their weak security if not deployed properly with security in mind. Similarly, the firmware update mechanism itself can also be a major attack vector if not designed appropriately. Due to the broadcast nature of the wireless medium, an adversary can launch both *external* and *insider* attacks. In *external* attacks, the attacker does not control any network node; however, the attacker can inject forged

packets, launch replay attacks, or even impersonate nodes. The attacker can also launch DOS attacks, injecting a vast amount of messages and exploiting the weakness of the dissemination protocol. In *insider* attacks, the adversary has managed to compromise a node and instructs it to intercept sensitive information, drop packets, inject false data, exploit weaknesses of the protocol, and so forth. Due to the epidemic nature of the update dissemination protocols, an adversary can gain complete control over the network by compromising a single node and then leveraging the reprogramming mechanism to distribute malicious code in every reachable node. For these reasons, the authenticity and integrity of the firmware image have to be verifiable by the network nodes. One way to achieve this is to sign the whole image using a suitable digital signature algorithm and verify it at the receiving node. However, this strategy requires the whole image to be received prior to verification, thus wasting valuable resources in case the received image has not been properly signed. Another option is to sign each different page of the image separately, or even to sign each network packet independently from the others. This would enable pipelining during dissemination and save network bandwidth but could increase processing cost at a receiving node, as digital signature verification should be performed for each individual page or packet. Several security-related firmware update contributions are discussed in the next sections (summarized in Table 4).

4.1 Selective 'n' Secure OTAP Protocol (SenSeOP)

The *SenSeOP* OTAP protocol [14] aims to secure the nodes of a wireless network from malicious and unauthorized reprogramming attempts, using an asymmetric cryptography approach based on the TinyECC library [70] that provides the necessary cryptographic operations for signature generation and verification. Similarly to *Deluge*, *SenSeOP* splits the update image into pages; however, it uses a simpler dissemination mechanism. In order to ensure the *integrity* and *authenticity* of a firmware update image, it leverages ECC digital signatures using 192-bit-length private keys. Moreover, using asymmetric cryptography, only the public key is stored in each node; hence, memory attacks are ineffective. The authors assume infrequent and non-regular software updates where confidentiality is not required, so no encryption is used. In order to save valuable resources, *SenseOP* computes the digital signature of the whole firmware image and not the signatures of the packets the image is subdivided to. This restricts the signature verification at the receiving side to be performed only when the whole image is received. Initially, the firmware update image is hashed and encrypted using the operator's private key to produce the corresponding signature. Afterward, the image is fragmented and the corresponding packets are transmitted through the network; broadcast, as well as unicast, transmission is possible. Replay attacks are avoided through the use of a version counter combined with the destination (or broadcast) address that is included in the hash computation as well.

4.2 Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks (Seluge)

Seluge [52] is a secure extension for *Deluge*, which provides integrity assurance for the firmware image and resistance against DoS attacks that specifically target firmware dissemination protocols. *Seluge* provides immediate authentication of each individual packet upon receipt, defeating the DoS attacks that exploit the authentication delays the node faces when waiting to receive the rest of a page. Moreover, all advertisements and requests are authenticated using a weak authentication scheme along with a signature, since it can be efficiently verified by a node; it still takes a vast amount of time for an attacker to forge the authenticator. The firmware image is split into fixed-size pages and each page is further divided into a number of packets. For every packet of a page P, a hash value is computed that is appended to the corresponding packet of page $P-1$. This is an iterative

Table 4. Security-enabled OTAP Protocols

Protocol	Confidentiality	Integrity	Authenticity	Digital Signature Scheme	Protection Against
SenSeOP [14]	No	Yes	Yes	SHA-1 & ECC	DoS, replay attacks
Seluge [52]	No	Yes	Yes	ECDSA	DoS, integrity, insider attacks
Sluice [64]	No	Yes	Yes	ECDSA	Integrity, insider attacks
Securing deluge [34]	No	Yes	Yes	SHA-1, RSA	Integrity, insider attacks
Secure firmware updates using open standards [106]	No	Yes	Yes	ed25519, EdDSA	Integrity attacks
ASSURED [15]	Yes	Yes	Yes	ed25519, EdDSA	Integrity, insider attacks
Secure FOTA object [27]	Yes	Yes	Yes	-	Integrity attacks

process, forming a hash tree [92] that is used as the basis for the computation of the final signature. For the digital signatures, the ECDSA algorithm is used over the 160-bit elliptic curve *secp160k1*. To disseminate the image, the source node first broadcasts the signature packet that serves as an advertisement for the new firmware image. Upon reception of this packet, a node cryptographically verifies it and then uses the root of the Merkle hash tree to authenticate each hash packet of the first page. Next, since the packets of the first page carry the hash values of those of the second page, signature verification continues, and this is repeated till all pages are verified. Nevertheless, this signature scheme is vulnerable to DoS attacks as an adversary can inject bogus signature packets and force the nodes to perform energy- and time-consuming operations. To address this issue, *Seluge* uses a weak authentication mechanism, called *message specific puzzles* [76], authenticating the signature packet for a specific firmware version with a key that is cryptographically associated with the version identity number. If verification succeeds, the node will verify the puzzle solution and authenticate the source of the signature packet. This way, the node will not perform any operation for bogus signature packets. A disadvantage of *Seluge* is that it increases the ROM and RAM utilization, as metadata and hash values have to be stored.

4.3 Secure Dissemination of Code Updates in Sensor Networks (Sluice)

Sluice [64] is based on *Deluge* and uses *hash chains* to ensure the authentication and integrity of the received firmware image, while providing pipeline support. Similarly to *Deluge*, *Sluice* splits each firmware image into several pages and integrates signatures and hash functions for efficient code authentication. More specifically, the hash image of each page is computed and is appended to the previous page, forming a chain of hashes. Following the concept of digital stream signing, the head of the hash chain (first page in the chain) only is signed using the private key of the firmware provider, requiring only one signature to be computed for the whole update. Using the digital signature, a node can verify the source of the update and can also ensure the integrity of the image, comparing the hash found in a received page with the computed hash value of the previous one; hence, using *Sluice*, there is a minimal overhead, as the only operations required are the signature validation and the computation of a few hash values. For the digital signature, the ECDSA algorithm is used with 160-bit SHA-1 hashes.

4.4 Securing Deluge

In [34], the authors propose a secure version of *Deluge* that utilizes authenticated digital streams. A firmware image is transformed into a series of messages, each one containing the hash of the previous message. The head of this hash chain is signed using RSA signatures and 64-bit SHA-1 hashes. In more detail, the firmware image is split into pages, where each page is further divided into a group of packets, and a hash value is computed for every such packet that is attached to the data of its previous packet. Finally, the hash value of the first packet is digitally signed, producing the signature, and these two values form the *advertisement packet* of the update, the first packet broadcasted when a new firmware version becomes available. If there is a receiving node that wishes for a firmware update, it checks the signature and caches the hash value of the first packet. Afterward, the normal dissemination methodology of *Deluge* is followed to request the packets of the first page. In order to ensure that packets arrive in order, the hash of each received packet is compared with the one that was stored in the last accepted one. If this comparison fails, the node requests a retransmission with a selective NACK message; hence, the packets should be received in order so that they can be accepted as legitimate packets of the firmware image, but out-of-order packets are also cached for optimization purposes.

4.5 Secure Firmware Updates Using Open Standards

The authors in [106] propose a firmware update mechanism based on open standards such as CoAP, LwM2M, SUIT, and so forth. The firmware server signs the firmware image and its metadata (manifest) using ECC and, more specifically, the *ed25519* and *ECDSA*/*p256r1* algorithms and elliptic curves. There is a two-process approach that gives higher flexibility: first only the metadata are transmitted, and if successfully verified by the receiving node (using a trust anchor with knowledge of the firmware provider's public key), the firmware image is downloaded using CoAP block operations. No deltas are used, and during transmission, neither the metadata nor the firmware image is encrypted.

4.6 Secure Software Update of Realistic Embedded Devices (ASSURED)

The authors in [15] propose a scalable architecture for OTAP, supporting end-to-end security. They distinguish four types of stakeholders: (1) **original equipment manufacturer (OEM)**, (2) firmware distributor, (3) domain controller, and (4) connected devices. *OEM* cryptographically signs a new firmware version using ECC, based on *Ed25519*, and the devices verify the signature prior to installing this new version. The firmware distributor's role is solely firmware distribution and can be a non-trusted entity, while the domain controller can set policies on the firmware update process (e.g., use of firmware deltas) that are included within a metadata structure (manifest). *ASSURED* was built in two proof-of-concept implementations: (1) on Hydra [35], a hybrid (HW/SW) remote attestation design based on a micro-kernel, which offers process memory isolation and enforces access control to memory regions, and (2) on ARM Cortex-M23 MCU that is equipped with Trustzone security extensions [84] and is able to partition the system into two regions (secure, non-secure).

4.7 Secure FOTA Object for IoT

In [27], the authors propose the use of a standardized approach and structure, commonly referred as *objects* that can be used by IoT manufacturers. This secure object is called *FOSE*. The motivation is that the packets that compose the firmware update image are usually secured by application layer security. However, the connectivity over which the transmission takes place usually breaks and it is impossible to resume later. *FOSE* ensures that no tampering of data will take place and

broken connections can be resumed in a later stage. In order to be able to resume the connection, the client ACKs each received a *FOSE object* and the server keeps track of the counter. The data contained inside a *FOSE object* are encrypted, thus providing confidentiality.

5 PLATFORMS SUPPORTING FIRMWARE OVER-THE-AIR PROGRAMMING

In this section, we present a collection of cloud platforms that offer firmware OTAP for IoT devices. Some of them are part of a broader IoT ecosystem that may support complex application domains, while others are exclusively focused on IoT device management, the OTAP software update being part of it. All information presented here originates mainly from the documentation provided for each platform. We focus on platforms that aim at providing robust, reliable, and secure OTAP. This is guaranteed by characteristics, such as atomic software installation, easy rollback to previous software version (e.g., through A/B update that alternates two slots/partitions for loading and storing the new software), update failure management (i.e., in case of power or connectivity loss), short downtime, secure communication during software downloading, and authenticity and integrity verification of new software. Table 5 summarizes the main aspects of the platforms under consideration.

5.1 Mender

Mender [7] is an open source over-the-air software update manager for embedded Linux devices, which considers security and reliability of the update process, and both application and full system updates are possible. *Mender* architecture is essentially built on two components: (1) *Mender Management Server* and (2) *Mender Client*. *Mender Management Server* is the central point for deploying updates to IoT devices. It monitors the software that is installed on each registered device and schedules the roll-out of new releases. Devices can be organized into groups, so that batch software updates can be orchestrated. *Mender Client* runs on the device and periodically polls the *Mender Management Server* for monitoring reasons (e.g., status reporting), as well as for discovering pending software updates. In case software updates exist for the specific device, *Mender Client* is responsible for downloading and installing it. Obviously, a software build system is necessary for generating new device software. Mender uses *Yocto Project* [10] for building the artifacts required by the target device. *Yocto Project* is a Linux Foundation collaborative open source project that offers tools and processes for creating custom embedded Linux distributions. It follows a layer model for developing logically independent software pieces that can be easily customized, combined, and reused, supporting its architecturally agnostic nature (it supports all major embedded architectures, such as ARM, 32-bit and 64-bit x86, PowerPC, and MIPS). Mender provides *meta-mender*, a set of Yocto Project layers for embedding *Mender Client* into the OS image. Reliability of the update process is enhanced by using dual rootfs updates (new software is deployed in inactive partition that becomes active after reboot), sanity checks during first reboot, and rollback to former software if the sanity checks fail. The *Mender* platform considers device authentication, software authenticity, and integrity, as well as secure communication between the *Mender Client* and the Mender Management Server. Initially, each device authenticates to the Management Server through an authentication set (identity attributes and public key). Subsequently, it is provided with an authentication token (JSON Web Token) by means of which each subsequent request is authenticated. Authenticity and integrity of the built software are guaranteed through cryptographic signatures that are verified at the device. Currently, the platform supports two algorithms, namely RSA with recommended key length of at least 3,072 bits and ECDSA with ECC `prime256v1` curve. Finally, communication between devices and back-end is secured through **Transport Layer Security (TLS)**. It is noted that although *Mender* is not a general-purpose IoT device management platform, as it is solely focused on managing and orchestrating the software updates, it has been

Table 5. Summary of Platforms Supporting OTAP

Platform	Mender [7]	ARM Pelion [3]	Balena [4]	Particle [9]	AWS IoT - FreeRTOS [5]
Supported processors	ARM, x86(-64)	ARM Cortex-M, ARM Cortex-A	ARM, x86(-64)	ARM Cortex-M	ARM Cortex-M, MIPS Warrior-M, Tensilica Xtensa LX6
Operating system	Embedded Linux (Yocto Project), Debian Family	ARM MbedOS	BalenaOS (built on Yocto Project)	Particle Device OS	Amazon FreeRTOS
Connectivity	WiFi, Cellular	WiFi, Cellular, BLE, IEEE 802.15.4, LoRa	WiFi, Cellular	WiFi, Cellular (2G, 3G, LTE), BLE, IEEE 802.15.4	WiFi, BLE
Device authentication	Public key, JSON Web Token	Public key, X.509 Certificate	API key	Public key	Public key, X.509 Certificate
Communication security	TLS	TLS, DTLS	TLS	DTLS, AES	TLS
Software/firmware authenticity and integrity	Cryptographic signatures (RSA, ECDSA)	Cryptographic signatures (ECDSA)	-	CRC32 (non-cryptographic)	Cryptographic signatures (RSA, ECDSA)
Update reliability and efficiency	– A/B update (Dual rootfs) – Update rollback – Batch update	– A/B update – Update rollback – Conditional update – Differential update – Continuous update – Batch update	– A/B update (Dual rootfs) – Update rollback (includes boot partition)	– A/B update – Update rollback – Context-aware update – Update on wake-up – Batch update	– A/B update – Update rollback – Context-aware update – Update on wake-up – Batch update

successfully integrated into other major IoT platforms, such as Google Cloud IoT Core and Microsoft Azure IoT.

5.2 ARM Pelion

ARM provides a full-stack IoT solution, spanning from embedded devices to IoT cloud services. The *ARM Pelion* IoT platform [3] is a suite of management services that focuses on three core IoT components, namely connectivity, device, and data management, for devices running the ARM Mbed OS or Linux/Mbed Linux OS. Several different processor architectures are supported, from simple Cortex-M microcontrollers to powerful Cortex-A systems, as well as both IP and non-IP-based communication protocols (e.g., LoRa, BLE), the latter with the support of an appropriate gateway that employs the necessary protocol translation. Communication between the IoT devices and the Management Server is based on the Open Mobile Alliance ***Lightweight Machine-to-Machine* (LwM2M)** application protocol that is used in combination with the ***Constrained Application***

Protocol **(CoAP)** over UDP or TCP transports. Transport layer security is provided by **Datagram Transport Layer Security (DTLS)** or TLS protocol, respectively. The LwM2M protocol provides a simple, yet very efficient, data model for (1) device bootstrapping, (2) discovery and registration, (3) device management and service enablement, and (4) information reporting. It is noted that the LwM2M data model provides native support for the firmware update process by defining the necessary resources the device needs to expose in order to receive the firmware binary. Over-the-air firmware updates are performed in the form of *campaigns* that apply either to a single device or fleet of devices. A typical full firmware image contains the OS, the *Device Management Update Client* (responsible for managing the update process on the device part), and the user application. Efficiency of the update process, especially for low-rate and low-power IoT devices (e.g., NB-IoT), is enhanced by delta updates (binary patches for constructing new firmware binary from the existing one). Reliability mechanisms include *conditional updates* (a device accepts updates based on pre-defined conditions, e.g., minimum battery level), *sanity checks* on received firmware (bootloader verifies firmware integrity by calculating its hash and comparing it with the one received as firmware metadata), and *rollback support* through dual partitions (active partition/candidate partition). Each firmware image is accompanied by a piece of information, named as *manifest*. The manifest is essentially firmware metadata that encodes information on firmware authenticity, integrity, device compatibility, and update logistics (update time scheduling, binary storage options, etc.).

ARM Pelion Device Management uses **Public Key Infrastructure (PKI)**-based security and relies on X.509 certificates and public-key encryption for server and device authentication. Authenticity and cryptographic integrity of a firmware binary and its corresponding manifest are achieved by ECDSA signatures (based on ECC `secp256r1` curve). In addition to the device management services, ARM provides an online compiler, named *Mbed Compiler*, for developing and building device firmware through a web SDK. The user can download the produced binary either locally or directly use Pelion services for performing a firmware OTA update campaign to registered devices. *Mbed Compiler* integrates features, such as source code version control and collaboration tools for multi-author projects. Finally, it hosts a large database of free user-created libraries that can be easily imported into any application.

5.3 Balena

Balena [4] offers a complete set of tools for building, deploying, and managing fleets of connected embedded Linux IoT devices. It builds on Linux containers technology for easily deploying updates on applications or even the entire host OS running on an IoT device. The main software components of the *Balena* ecosystem are described in the following. IoT devices run *Balena* OS, a Yocto Project Linux-based OS, packaged with *balenaEngine*, a lightweight Docker-compatible container engine that manages containers executing *Balena* services or user applications. On the cloud side, the *BalenaCloud* platform is responsible for device and communication management, as well as source code version control (though `git` repositories), software build, storage, and device update. A set of CLI tools is also provided for local development, local or remote software building, and device software deployment. A device authenticates to BalenaCloud by using API keys. During provisioning it receives a provisioning API key that is used for authentication after first boot-up and registration to the back-end. Then, a new API key is generated and provided to the device, which uses it for subsequent authentication. Device control and image download during updates is performed over a **Virtual Private Network (VPN)**, so that all traffic is secured with TLS. Reliability of the update process is based on pre-update sanity checks in terms of software-device compatibility and availability of the image in the container registry, double root partition approach (active/inactive partition), and rollback support, in case boot partition changes fail (remember that it is possible to update the entire host operating system, not only a user application that is executed on the device).

5.4 Particle

Particle [9] offers a full-stack IoT solution that includes different hardware platforms, various connectivity options and cloud services for device fleet management, over-the-air updates, and device health monitoring. Microprocessor architectures supported are of ARM Cortex-M series, while connectivity options include WiFi, Cellular (2G, 3G, LTE), BLE, and IEEE 802.15.4. Devices run the *Particle Device OS* operating system that provides the necessary hardware abstraction for easy application development and enables connectivity and management functionalities. On the other end, the *Device Cloud* is the cloud back-end that provides device management and monitoring through real-time event and data logging, firmware roll-outs, and over-the-air updates, as well as integration with other major IoT platforms (e.g., Google Cloud Platform, Azure IoT Hub) and data publishing through Webhooks. Mutual authentication between Device Cloud and the devices is based on RSA public/private key pairs. Communication is encrypted by using DTLS over UDP or AES over TCP (depending on the device capabilities), and CoAP is used for data collection and device management operations, including over-the-air firmware update. Reliability and resiliency of the firmware update process are supported by features such as *atomic updates*, according to which only a fully received and verified firmware is executed; *automatic rollbacks*, in case of failure during firmware transfer; *context-aware updates* based on device operational status (e.g., update is postponed for a later time, if the device currently performs a critical task); *update on wake-up*, for currently sleeping devices; and *batch updates*. It is noted that updates are possible at both an application and Device OS level and can be initiated through either provided CLI tools, Device Cloud Web UI, or Device Cloud REST API. The platform offers additionally a Web IDE, named *Pacticle Build,* and a database of libraries for developing and building device firmware in the cloud, which can be further released to registered device fleets.

5.5 AWS IoT - FreeRTOS

FreeRTOS [5] is a well-known preemptive real-time operating system for embedded devices that employs a priority-based dynamic scheduler and a multi-threading programming model. It is open source and has been successfully ported to a large number of microcontroller/microprocessor platforms, including ARM variants (ARM7, ARM9, Cortex-M, and Cortex-A Series), Atmel AVR, MSP430, Espressif ESP32, and so forth. An extension of FreeRTOS, provided by Amazon, includes necessary libraries for the secure and reliable connection of IoT devices with the ***Amazon Web Services IoT*** **(AWS-IoT)** cloud platform [2], for device management, data collection, and application development. IoT devices communicate with the cloud back-end through the **Message Queue Telemetry Transport (MQTT)** protocol, either directly or over Websockets. In addition, an HTTP REST endpoint exists for data publishing. Mutual authentication is achieved by using PKI X.509 certificates, and the TLS protocol is used for securing the communication channel between IoT devices and the AWS IoT back-end. Over-the-air firmware update is supported for devices running Amazon FreeRTOS. The *OTA Update Manager* service, which is part of the AWS IoT back-end, is responsible for notifying the device on existing updates, orchestrating the update process, and maintaining the update log. On the device side, the *OTA Agent* module manages the notification, downloading (over secure MQTT or HTTP), and verification of the firmware updates. In order to ensure the authenticity and integrity of the firmware, the OTA Update Manager cryptographically signs the firmware image before deployment (both RSA and ECDSA signatures are supported). The OTA Agent verifies the signature before applying any new update. In order to improve the reliability and efficiency of the update process, the AWS IoT for FreeRTOS provides support for *batch updates* (device group or entire fleet), *continuous updates* (so that new firmware is deployed to devices as they are added to groups, reset, or re-provisioned), and *update rollback*.

6 CONCLUSION-FURTHER WORK

The proliferation of massive IoT networks has been remarkable in the last years. These ubiquitous smart-object-enabled networks, which may operate for several years in variable conditions, are used for supporting complex applications in several domains, such as smart cities, healthcare, industrial automation, and so forth. Throughout their extended lifetime, the nodes forming the IoT networks need to be re-programmed, so that new features are added, software bugs or security vulnerabilities are resolved, and their functionality is re-purposed. The large scale of IoT networks and the usual installation of IoT nodes in locations with difficult or no physical access mandate the use of OTAP solutions for the efficient update of firmware running on IoT nodes. In this article, we presented an overview of OTAP techniques that can be applied to IoT networks. We highlighted the main challenges and limitations stemming from the resource-constrained and heterogeneous nature of IoT nodes and analyzed the essential stages of the firmware update process, along with different approaches and techniques that implement them. In addition, we discussed schemes that focus on securing the OTAP process by encrypting the transmitted firmware and/or providing firmware authenticity and integrity-preserving mechanisms. Finally, we presented a collection of state-of-the art commercial and open-source platforms that integrate secure and reliable OTAP.

OTAP has attracted a vast amount of interest from the research community and industry. As a result, future research in the field is quite a challenging task; however, we can recognize several OTAP aspects that can accept further contributions. First, the implementation of lightweight compression/decompression algorithms can enable the shrinkage of the firmware image that is transmitted to the target devices. Besides, such algorithms can be used along with differencing algorithms resulting in the substantial reduction of the transmissions during the update. We advocate that new differencing algorithms can be implemented utilizing modern data structures (e.g., suffix arrays), achieving both low execution time and small delta scripts. As modern communication protocols (e.g., LwM2M, CoAP, etc.) are increasingly integrated into resource-constrained networks, future contributions may build on top of them, providing pipelined data dissemination, and also guarantee the freshness, integrity, and authentication of the received data (delta script or complete firmware image).

REFERENCES

[1] 2006. Tmote Sky. Retrieved September 15, 2020, from https://insense.cs.st-andrews.ac.uk/files/2013/04/tmote-sky-datasheet.pdf.
[2] 2020. Amazon Web Services IoT. Retrieved September 15, 2020, from https://aws.amazon.com/iot.
[3] 2020. Arm Pelion IoT Platform. Retrieved September 15, 2020, from https://www.pelion.com.
[4] 2020. Balena - The complete IoT fleet management platform. Retrieved September 15, 2020, from https://www.balena.io.
[5] 2020. FreeRTOS - Real-time operating system for microcontrollers. Retrieved September 15, 2020, from https://www.freertos.org/.
[6] 2020. Map files GNU. Retrieved September 15, 2020, from https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html.
[7] 2020. Mender - Open source over-the-air software updates for Linux devices. Retrieved September 15, 2020, from https://mender.io/.
[8] 2020. MICAz. Retrieved September 15, 2020, from http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf.
[9] 2020. Particle. Retrieved September 15, 2020, from https://www.particle.io/.
[10] 2020. Yocto Project. Retrieved September 15, 2020, from https://www.yoctoproject.org/.
[11] I. Adly, H. F. Ragai, A. El-Hennawy, and K. A. Shehata. 2010. Over-the-air programming of PSoC sensor interface in wireless sensor networks. In *Proceedings of the Mediterranean Electrotechnical Conference (MELECON'10)*. 997–1002.
[12] S. Alagar, S. Venkatesan, and J. Cleveland. 1995. Reliable broadcast in mobile wireless networks. In *Proceedings of (MILCOM'95)*, Vol. 1. 236–240.

[13] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (Usenix'17)*. 1093–1110.

[14] N. Aschenbruck, J. Bauer, J. Bieling, A. Bothe, and M. Schwamborn. 2012. Selective and secure over-the-air programming for wireless sensor networks. In *2012 21st International Conference on Computer Communications and Networks (ICCCN'12)*. 1–6.

[15] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghu, and G. Tsudik. 2018. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 37 (2018), 2290–2300.

[16] E. Baccelli, J. Doerr, S. Kikuchi, F. Padilla, K. Schleiser, and I. Thomas. 2018. Scripting over-the-air: towards containers on low-end devices in the internet of things. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops'18)*. IEEE, 504–507.

[17] U. Banerjee, A. Wright, C. Juvekar, M. Arvind, and A. Chandrakasan. 2019. An energy-efficient reconfigurable DTLS cryptographic engine for securing internet-of-things applications. *IEEE Journal of Solid-state Circuits* 54 (2019), 2339–2352.

[18] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. De Poorter. 2020. Over-the-air software updates in the internet of things: An overview of key principles. *IEEE Communications Magazine* 58 (2020), 35–41.

[19] C. Bormann, M. Ersue, and A. Keranen. 2020. RFC 7228 - Terminology for Constrained-Node Networks. Retrieved September 15, 2020, from https://datatracker.ietf.org/doc/rfc7228/.

[20] S. Brown and C. Sreenan. 2013. Software updating in wireless sensor networks: a survey and lacunae. *Journal of Sensor and Actuator Networks* 2 (2013), 717–760.

[21] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. 1998. A digital fountain approach to reliable distribution of bulk data. *ACM SIGCOMM Computer Communication Review* 28 (1998), 56–67.

[22] D. Chen, D. He, and F. Ahmad. 2016. A survey of reprogramming security in wireless sensor network. *VFAST Transactions on Software Engineering* 4 (2016), 7–20.

[23] X. Cheng, M. Ding, D. Du, and X. Jia. 2006. Virtual backbone construction in multihop ad hoc wireless networks. *Wireless Communications and Mobile Computing* 6 (2006), 183–190.

[24] A. Chlipala, J. Hui, and G. Tolle. 2004. Deluge: Data dissemination for network reprogramming at scale. Class Project, Berkeley, University of California.

[25] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. 2008. Better external memory suffix array construction. *Journal of Experimental Algorithmics* 12 (2008), 1–24.

[26] B. Djamaa and M. Richardson. 2015. Optimizing the trickle algorithm. *IEEE Communications Letters* 19, 5 (2015), 819–822.

[27] K. Doddapaneni, R. Lakkundi, S. Rao, S. G. Kulkarni, and B. Bhat. 2017. Secure FoTA object for IoT. In *2017IEEE 42nd Conference on Local Computer Networks Workshops (LCN Workshops)*. 154–159.

[28] C. Dong and F. Yu. 2014. An efficient network reprogramming protocol for wireless sensor networks. *Computer Communications* 55 (2014), 41–50.

[29] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao. 2013. R2: incremental reprogramming using relocatable code in networked embedded systems. *IEEE Transactions on Computers* 62 (2013), 1837–1849.

[30] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen. 2010. Elon: Enabling efficient and long-term reprogramming for wireless sensor networks. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* SIGMETRICS*(SIGMETRICS'10)*. ACM Press, 49.

[31] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen. 2013. R3: optimizing relocatable code for efficient reprogramming in networked embedded systems. In *Proceedings of the IEEE INFOCOM (INFOCOM'13)*. 315–319.

[32] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference onEmbedded Networked Sensor Systems* SenSys*(SenSys'06)*. ACM Press, 15.

[33] A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*. 455–462.

[34] P. Dutta, J. Hui, D. Chu, and D. Culler. 2006. Securing the deluge network programming system. In *IPSN*.

[35] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. 2017. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Proceedings of ACM WiSec (WiSec'17)*. 99–110.

[36] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. 2003. EmStar: an environment for developing wireless embedded systems software. Technical Report.

[37] M. Ersue, D. Romascanu, J. Schoenwaelder, and A. Sehgal. 2015. Management of networks with constrained devices: Use cases, RFC7548. Technical Report.

[38] M. Farooq and T. Kunz. 2011. Operating systems for wireless sensor networks: A survey. *Sensors* 11 (2011), 5900–5930.
[39] A. Fragkiadakis and E. Tragos. 2016. A trust-based scheme employing evidence reasoning for iot architectures. In *IEEE 3rd World Forum on Internet of Things (WF-IoT'16)*. 559–564.
[40] E. Frimpong and A. Michalas. 2020. IoT-CryptoDiet: Implementing a Lightweight Cryptographic Library Based on ECDH and ECDSA for the Development of Secure and Privacy-Preserving Protocols in Contiki-NG. In *5th International Conference on Internet of Things, Big Data and Security*. 101–111.
[41] D. Frisch, S. Reißmann, and C. Pape. 2017. An over the air update mechanism for ESP8266 microcontrollers. In *the Twelfth International Conference on Systems and Networks Communications, 2017, 12-17.*
[42] B. Ghaleb, A. Al-Dubai, and E. Ekonomou. 2015. E-trickle: Enhanced trickle algorithm for low-power and lossy networks. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 1123–1129.
[43] S. Guha and S. Khuller. 1998. Approximation algorithms for connected dominating sets. *Algorithmica* 20 (1998), 374–387.
[44] A. Hagedorn, D. Starobinski, and A. Trachtenberg. 2008. Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes. In *2008 International Conference on Information Processing in Sensor Networks (IPSN'08)*. 457–466.
[45] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. 2016. Operating systems for low-end devices in the internet of things: A survey. *IEEE Internet of Things Journal* 3 (2016), 720–734.
[46] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. 2005. SOS -a dynamic operating system for sensor networks. In *Proceedings of Mobisys (Mobisys'05)*.
[47] J. Hill and D. Culler. 2002. Mica: A wireless platform for deeply embedded networks. *IEEE Micro* 22, 6 (2002), 12–24.
[48] D. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18 (1975), 341–343.
[49] T. Ho, M. Medard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. 2006. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory* 52 (2006), 4413–4430.
[50] J. Hu, Chun Jason Xue, Yi He, and Edwin H.-M. Sha. 2009. Reprogramming with minimal transferred data on wireless sensor network. In *Proceedings of the 6th International Conference on Mobile Adhoc and Sensor Systems (MASS'09)*. IEEE, 160–167.
[51] L. Huang and S. Setia. 2008. CORD: Energy-efficient reliable bulk data dissemination in sensor networks. In IEEE INFOCOM 2008 - The 27th Conference on Computer Communications. 574–582.
[52] S. Hyun, P. Ning, A. Liu, and W. Du. 2008. Seluge: Secure and dos-resistant code dissemination in wireless sensor networks. In *2008 International Conference on Information Processing in Sensor Networks (IPSN'08)*. 445–456.
[53] Jaein J. and D. Culler. 2004. Incremental network programming for wireless sensors. In *Proceedings of the 1st Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON'04)*. 25–33.
[54] J. Jeong. 2003. Node-level Representation and System Support for Network Programming.
[55] O. Kachman. 2016. Configurable Reprogramming Scheme for Over-the Air Updates in Networked Embedded Systems. http://www.fit.vutbr.cz/events/pad2016/download/sbornik/11-Kachman.pdf. Accessed: 2020-09-15.
[56] O. Kachman. 2018. Effective Multiplatform Firmware Update Process for Embedded Low-Power Devices. Retrieved September 15, 2020, from http://acmbulletin.fiit.stuba.sk/vol11num1/kachman2019.pdf.
[57] O. Kachman and M. Balaz. 2016. Optimized differencing algorithm for firmware updates of low-power devices. In *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'16)*. IEEE, 1–4.
[58] A. Kishore. 2017. Turning internet of things (IoT) into internet of vulnerabilities (IoV): IoT botnets. Retrieved September 15, 2020, from https://arxiv.org/abs/1702.03681v1.
[59] D. Korn, J. MacDonald, J. Mogul, and K. Vo. 2002. The VCDIFF Generic Differencing and Compression Data Format. RFC Editor. Published: RFC 3284.
[60] J. Koshy and R. Pandey. 2005. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks, 2005 (EWSN'05)*. IEEE, 354–365.
[61] S. Kulkarni and M. Arumugam. 2006. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal of Distributed Sensor Networks* 2 (2006), 55–78.
[62] S. Kulkarni and L. Wang. 2005. MNP: Multihop network reprogramming service for sensor networks. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. 7–16.
[63] V. Lakkundi and K. Singh. 2014. Lightweight DTLS implementation in CoAP-based internet of things. In *Proceedings of ADCOM (ADCOM'14)*.
[64] P. Lanigan, R. Gandhi, and P. Narasimhan. 2006. Sluice: Secure dissemination of code updates in sensor networks. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*.

[65] K. Lehniger and S. Weidling. 2019. The impact of diverse execution strategies on incremental code updates for wireless sensor networks:. In *Proceedings of the 8th International Conference on Sensor Networks*. SCITEPRESS - Science and Technology Publications, 30–39.

[66] P. Levis, N. Lee, M. Welsh, and D. Culler. 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*.

[67] P. Levis, N. Patel, D. Culler, and S. Shenker. 2004. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation* - Volume 1 *(NSDI'04)*. USENIX Association.

[68] L. Li, Z. Xi, Y. Zhu, and S. Wang. 2018. Improvement and implementation of RPL routing protocol in wireless sensor networks. In *WiCOM*.

[69] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. 2017. A survey on Internet of Things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal* 4 (2017), 1125–1142.

[70] A. Liu and P. Ning. 2008. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In *2008 International Conference on Information Processing in Sensor Networks (IPSN'08)*. 245–256.

[71] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. 2004. Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSYS'04)*. ACM Press, 256.

[72] B. Mazumder and J. O. Hallstrom. 2013. An efficient code update solution for wireless sensor network reprogramming. In *Proceedings of the 2013 International Conference on Embedded Software (EMSOFT'13)*. 1–10.

[73] B. Mo, W. Dong, C. Chen, J. Bu, and Q. Wang. 2012. An efficient differencing algorithm based on suffix array for reprogramming wireless sensor networks. In *Proceedings of the 2012 IEEE International Conference on Communications (ICC'12)*. 773–777.

[74] M. Mughal, X. Luo, A. Ullah, S. Ullah, and Z. Mahmood. 2018. A lightweight digital signature based security scheme for human-centered internet of things. *IEEE Access* 6 (2018), 31630–31643.

[75] V. Naik, A. Arora, P. Sinha, and Hongwei Z. 2005. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*. 777–789.

[76] P. Ning, A. Liu, and W. Du. 2008. Mitigating dos attacks against broadcast authentication in wireless sensor networks. *ACM Transactions on Sensor Networks* 4 (2008), 1–35.

[77] J. Pallister, K. Eder, S. Hollis, and J. Bennett. 2014. A high-level model of embedded flash energy consumption. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'14)*. Association for Computing Machinery.

[78] R. Panta and S. Bagchi. 2009. Hermes: fast and energy efficient incremental code updates for wireless sensor networks. In *IEEE INFOCOM 2009 - The 28th Conference on Computer Communications (INFOCOM'09)*. IEEE, 639–647.

[79] R. Panta, S. Bagchi, and S. Midkiff. 2009. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *USENIX*.

[80] R. Panta, I. Khalil, and S. Bagchi. 2007. Stream: low overhead wireless reprogramming for sensor networks. In *Proceedings of IEEE INFOCOM (INFOCOM'07)*. 928–936.

[81] H. Park, J. Jeong, and P. Mah. 2014. Non-invasive rapid and efficient firmware update for wireless sensor networks. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing Adjunct Publication (UbiComp'14 Adjunct)*. ACM Press, 147–150.

[82] S. Park, R. Vedantham, R. Sivakumar, and I. Akyildiz. 2004. A scalable approach for reliable downstream data delivery in wireless sensor networks. In *Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'04)*.

[83] R. Parthasarathy, N. Peterson, W. Song, A. Hurson, and B. Shirazi. 2010. Over the air programming on Imote2-Based sensor networks. In *2010 43rd Hawaii International Conference on System Sciences (HICSS'10)*. 1–9.

[84] S. Pinto and N. Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *Computing Surveys* 51 (2019), 1–36.

[85] Prabal D., M. Grimmer, A. Arora, S. Bibyk, and D. Culler. 2005. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Fourth International Symposium on Information Processing in Sensor Networks, 2005 (IPSN'05)*. 497–502.

[86] Qiang W., Yaoyao Z., and L. Cheng. 2006. Reprogramming wireless sensor networks: Challenges and approaches. *IEEE Network* 20, 3 (2006), 48–55.

[87] N. Reijers and K. Langendoen. 2003. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications (WSNA'03)*. Association for Computing Machinery, 60–67.

[88] E. Ronen, A. Shamir, A. Weingarten, and C. OFlynn. 2017. IoT goes nuclear: Creating a Zigbee chain reaction. In *Proceedings of IEEE Symposium on Security and Privacy (Euro S&P'17)*.
[89] M. Sanvido, F. R. Chu, A. Kulkarni, and R. Selinger. 2008. Nand flash memory and its role in storage architectures. *Proceedings of the IEEE* 96 (2008), 1864–1874.
[90] N. Shafi, K. Ali, and H. Hassanein. 2012. No-reboot and zero-flash over-the-air programming for wireless sensor networks. In *Proceedings of the 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'12)*. 371–379.
[91] J. Shi, J. Wan, H. Yan, and H. Suo. 2011. A survey of cyber-physical systems. In *Proceedings of the International Conference on Wireless Communications and Signal Processing (WCSP'11)*. 1–6.
[92] A. Shoufan and N. Huber. 2010. A fast hash tree generator for Merkle signature scheme. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*. 3945–3948.
[93] F. Stann and J. Heidemann. 2003. RMST: Reliable data transport in sensor networks. In *Proceedings of the 1st IEEE International Workshop on Sensor Network Protocols and Applications (SNPA'03)*. 102–112.
[94] T. Stathopoulos, J. Heidemann, and D. Estrin. 2004. A Remote Code Update Mechanism for Wireless Sensor Networks. Technical Report.
[95] M. Stolikj, P. J. Cuijpers, and J. Lukkien. 2013. Efficient reprogramming of wireless sensor networks using incremental updates. In *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops'13)*. 584–589.
[96] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. 2004. Lessons from a sensor network expedition. In *EWSN*.
[97] A. Tal. 2002. White paper two flash technologies compared: NOR vs NAND.
[98] Crossbow Technology. 2003. Mote In-Network Programming User Reference Version 20030315. Crossbow Technology, Inc.
[99] E. Tragos, A. Fragkiadakis, V. Angelakis, and H. Pohls. 2017. Designing secure iot architectures for smart city applications. In *Designing, Developing, and Facilitating Smart Cities: Urban Design and IoT Solutions*. Springer.
[100] A. Tridgell. 1999. *Efficient Algorithms for Sorting and Synchronization*. Ph.D. Dissertation. The Australian National University.
[101] Y. Tseng, S. Ni, Y. Chen, and J. Sheu. 2002. The broadcast storm problem in a mobile ad hoc network. *Wireless Networks* 8 (2002), 153–167.
[102] B. Wang, Y. Chen, H. Gu, J. Yang, and T. Zhao. 2005. Two energy-efficient, timesaving improvement mechanisms of network reprogramming in wireless sensor network. In *Embedded Software and Systems*. Springer, Berlin, 473–483.
[103] Y. Wang, G. Attebury, and B. Ramamurthy. 2006. A survey of security issues in wireless sensor networks. *IEEE Communications Surveys Tutorials* 8 (2006), 2–23.
[104] C. Wilson. 1999. Sensors in medicine. *Western Journal of Medicine* 171 (1999), 322–325.
[105] J. Yiu. 2015. Introduction to embedded software development. In *The Definitive Guide to Armő Cortexő-M0 and Cortex-M0+ Processors*. Elsevier, 55–86.
[106] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli. 2019. Secure firmware updates for constrained iot devices using open standards: A reality check. *IEEE Access* 7 (2019), 71907–71920.
[107] C. Zhang, Wonsun Ahn, Youtao Zhang, and Bruce R. Childers. 2016. Live code update for IoT devices in energy harvesting environments. In *Proceedings of the 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA'16)*. IEEE, 1–6.
[108] H. Zhu, Z. Zhang, J. Du, S. Luo, and Y. Xin. 2018. Detection of selective forwarding attacks based on adaptive learning automata and communication quality in wireless sensor networks. *International Journal of Distributed Sensor Networks* 14 (2018), 1–15.

Received September 2020; accepted June 2021