Multiplatform incremental OTAP strategy for resource-constrained IoT devices

Konstantinos Arakadakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Xenofontas Dimitropoulos*

Thesis Co-Advisor: Dr. *Alexandros Fragiadakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This research has been 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).

University of Crete
Computer Science Department

Multiplatform incremental OTAP strategy for resource-constrained IoT devices

Thesis submitted by
Konstantinos Arakadakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Konstantinos Arakadakis

Committee approvals: ______________________
Xenofontas Dimitropoulos
Professor, Thesis Supervisor

____
Kostas Magoutis
Associate Professor, Committee Member

____
Hamed Haddadi
Professor, Committee Member

Departmental approval: ______________________
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, July 2021

Multiplatform incremental OTAP strategy for resource-constrained IoT devices

Abstract

The Internet of Things (IoT) presents itself as an emerging technology, which is able to interconnect a soaring number of heterogeneous smart devices around the world, for supporting complex data-driven applications in a variety of domains. The ability to wirelessly update these devices is paramount, as it allows the integration of additional functionality into their firmware, the resolution of code errors and security vulnerabilities, or even their complete re-purpose, without physically accessing them. Such *Over-the-Air Programming* (*OTAP*) solutions require the compaction of the transmitted data during a network update, in order to reduce the energy consumption in the network, which is due to the necessary communication operations.

To meet this need, IoT devices can also be updated using a technique, called *incremental programming*, that avoids the retransmission of the entire firmware, when an updated version has been released. The most common form of this technique is through the so-called differencing algorithms, that execute at a firmware server and aim to detect common segments between two firmware versions, producing an encoded patch that is disseminated to the network. This way, the devices can utilise parts of the firmware they currently run, in order to reconstruct the updated version locally.

In this work, we survey techniques, protocols, and schemes that focus on the reduction of the transmissions during a network update, and have found widespread application in large-scale IoT networks. We also emphasize the essential steps of the firmware update process, along with the proposed approaches and techniques that implement them. Besides, we discuss contributions that focus on the security aspects of OTAP, as recent cyberattacks have revealed that unsecured update strategies can allow adversaries to inject faulty or malicious firmware into a network, manipulating its operation.

Furthermore, we present a differencing algorithm we have developed, called *Dfinder*, that can compute small patches, based on delta encoding. The algorithm has $O(nlogn)$ time and $O(n)$ space complexity, and utilises *enhanced suffix arrays* and state-of-the-art construction techniques, that enable the efficient detection of common segments between two firmware versions. Additionally, we propose an extension of the algorithm, that halves the storage requirements at the receiver, so that devices with limited storage can also be updated, incrementally. Finally, we integrate *Dfinder* in an OTAP testbed and show that it can reduce the update time and the corresponding energy consumption of IoT networks up to 96%.

Πολυπλατφορμική στρατηγική για την επαυξητική ασύρματη ενημέρωση συσκευών διαδικτύου-των-πραγμάτων περιορισμένων πόρων

Περίληψη

Το διαδίκτυο των πραγμάτων (IoT) αποτελεί μια εξελισσόμενη τεχνολογία, η οποία είναι σε θέση να διασυνδέσει έναν αυξανόμενο αριθμό έξυπνων συσκευών, για την υποστήριξη πολύπλοκων δεδομενοκεντρικών εφαρμογών. Η δυνατότητα ασύρματης αναβάθμισης αυτών των συσκευών είναι εξέχουσας σημασίας, καθώς επιτρέπει την ενσωμάτωση επιπλέον λειτουργικότητας στο λογισμικό τους, τη διόρθωση λαθών και ευπαθειών του κώδικα, ή ακόμα και τον επαναπροσδιορισμό του σκοπού τους, δίχως να απαιτείται η φυσική επαφή με αυτές. Τέτοιες λύσεις ασύρματου προγραμματισμού, απαιτούν τη σύμπτυξη των αποσταλέντων δεδομένων κατά την αναβάθμιση, ώστε να μειωθεί η κατανάλωση ενέργειας στο δίκτυο, η οποία οφείλεται στις απαραίτητες λειτουργίες ασύρματης επικοινωνίας.

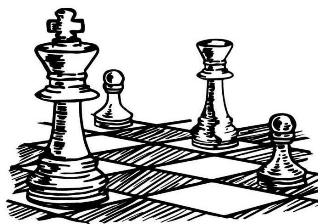
Για να ικανοποιηθεί αυτή η ανάγκη, οι συσκευές IoT μπορούν επίσης να αναβαθμιστούν, μέσω μιας τεχνικής, που ονομάζεται *επαυξητικός προγραμματισμός*, η οποία αποφεύγει την επαναποστολή ολόκληρου του λογισμικού, όταν έχει δημιουργηθεί μια ανανεωμένη έκδοση. Η πιο διαδεδομένη μορφή αυτής της τεχνικής, είναι μέσω των αλγόριθμων διαφοράς, οι οποίοι βρίσκουν τα κοινά τρήματα δύο εκδόσεων λογισμικού και υπολογίζουν ένα κωδικοποιημένο "μπάλωμα", το οποίο αποστέλλεται στο δίκτυο. Με αυτόν τον τρόπο, οι συσκευές μπορούν να χρησιμοποιήσουν μέρη του λογισμικού που τρέχουν εκείνη τη στιγμή, ώστε να συνθέσουν τη νέα έκδοση τοπικά.

Σε αυτήν την εργασία, μελετάμε τεχνικές, πρωτόκολλα δικτύου και συστήματα που έχουν αναπτυχθεί με σκοπό τη μείωση του όγκου των μεταδόσεων κατά την αναβάθμιση δικτύων IoT και έχουν βρει απήχηση σε δίκτυα μεγάλης κλίμακας. Επιπλέον, δίνουμε έμφαση στην ανάλυση των βημάτων της διαδικασίας αναβάθμισης, παρουσιάζοντας διάφορες τεχνικές που τα υλοποιούν. Ακόμη, αναφέρουμε μηχανισμούς οι οποίοι επικεντρώνονται στην ασφάλεια του ασύρματου προγραμματισμού, μιας και πρόσφατες επιθέσεις έχουν αποκαλύψει πως ανασφαλείς στρατηγικές αναβάθμισης, μπορούν να επιτρέψουν την εγκατάσταση επιβλαβούς λογισμικού από τις συσκευές ενός δικτύου, επηρεάζοντας τη λειτουργία τους.

Επιπλέον, παρουσιάζουμε έναν αλγόριθμο διαφοράς, ονόματι *Dfinder*, ο οποίος μπορεί να υπολογίσει μικρά "μπαλώματα", χρησιμοποιώντας κωδικοποίηση δέλτα. Ο αλγόριθμος έχει $O(nlogn)$ χρονική και $O(n)$ χωρική πολυπλοκότητα και χρησιμοποιεί επαυξημένους πίνακες αποθεμάτων και σύγχρονες τεχνικές υπολογισμού τους, που επιτρέπουν την αποδοτική εύρεση κοινών τμημάτων μεταξύ δύο εκδόσεων λογισμικού. Ακόμη, παρουσιάζουμε μία επέκταση του αλγόριθμου, η οποία μειώνει τις αποθηκευτικές απαιτήσεις στη μεριά του παραλήπτη κατά το ήμισυ, ώστε συσκευές με περιορισμένο αποθηκευτικό χώρο να μπορούν επίσης να αναβαθμιστούν επαυξητικά. Τέλος, ενσωματώνουμε το *Dfinder* σε μία πλατφόρμα ασύρματου προγραμματισμού δικτύων IoT και δείχνουμε πως μπορεί να μειώσει τον χρόνο και την ενέργεια που απαιτούνται για την αναβάθμιση δικτύων IoT μέχρι και 96%.

Ευχαριστίες

Θα ήθελα να εκφράσω τη βαθιά εκτίμηση και ευγνωμοσύνη μου στον επόπτη καθηγητή μου κύριο Ξενοφώντα Δημητρόπουλο, ο οποίος με φιλοξένησε στην ερευνητική του ομάδα, καθ΄ όλη τη διάρκεια των μεταπτυχιακών μου σπουδών. Μέσα σε αυτό το περιβάλλον κατάφερα να ανταλλάξω ιδέες και γνώσεις, οι οποίες επέτρεψαν την εξέλιξή μου τόσο σαν επιστήμονας όσο και σαν άνθρωπος. Ακόμη, οφείλω να τον ευχαριστήσω διότι με την βοήθεια του κατάφερα να μείνω και να εργαστώ για πέντε μήνες στη Σιγκαπούρη, κάνοντας πρακτική άσκηση στο National University of Singapore (NUS) , όπου αποκόμισα σημαντικές γνώσεις και εμπειρίες. Επιπλέον, δεν θα μπορούσα να ευχαριστήσω αρκετά τον συνεπιβλέποντα ερευνητή Αλέξανδρο Φραγκιαδάκη, ο οποίος υπομονετικά με βοήθησε κατά την διάρκεια αυτής της εργασίας, προφέροντας μου υλικό, γνώσεις και προτρέποντάς με να βελτιώνομαι συνεχώς. Τέλος, οφείλω να ευχαριστήσω τους γονείς μου, Καλλιόπη και Μανώλη, οι οποίοι πίστεψαν σε μένα και με στήριξαν με όποιο τρόπο μπορούσαν στις σπουδές μου, όπως και τα υπόλοιπα μέλη της οικογένειας μου, που ήταν πάντα πρόθυμοι να με ακούσουν και να μου δώσουν μια συμβουλή. Ένας εξ΄ αυτών είναι και ο παππούς μου, ο Μιχάλης, ο οποίος έφυγε στις αρχές του μεταπτυχιακού μου, αλλά γνωρίζω πόσο περήφανος θα ένιωθε.



Στην οικογένειά μου

Table of contents

Table of contents i

List of Tables v

List of Figures vii

1 Introduction 1
- 1.1 Internet of Things (IoT) . . . 1
  - 1.1.1 Wireless Sensor Networks (WSNs) . . . 2
- 1.2 Over-The-Air Programming (OTAP) . . . 3
- 1.3 Problem Statement . . . 4
- 1.4 Incremental Programming . . . 5
- 1.5 Contributions . . . 7

2 Related Literature 9
- 2.1 Improving Firmware Similarity . . . 10
  - 2.1.1 Slop Regions . . . 12
  - 2.1.2 Position-Independent Code (PIC) . . . 13
  - 2.1.3 Indirection Tables . . . 13
  - 2.1.4 Relocatable Code . . . 14
- 2.2 Differencing Algorithms . . . 14
  - 2.2.1 Fixed Block Comparison (FBC) . . . 15
  - 2.2.2 Rsync . . . 15
  - 2.2.3 Xdelta . . . 16
  - 2.2.4 BSDiff . . . 16
  - 2.2.5 RMTD . . . 17
  - 2.2.6 DASA . . . 17
  - 2.2.7 R3diff . . . 18
  - 2.2.8 Delta Generator (DG) . . . 18
- 2.3 Dissemination Protocols . . . 19
  - 2.3.1 XNP . . . 20
  - 2.3.2 Trickle . . . 20
  - 2.3.3 Deluge . . . 20
  - 2.3.4 Multihop Network Reprogramming (MNP) . . . 21

2.3.5 Seluge 21
2.3.6 Sluice 22
2.3.7 Secure firmware updates using open standards 22
2.3.8 ASSURED 22
2.4 Comparison of the Related Literature and our Contribution 23

3 Enhanced Suffix Arrays and Applications 25
3.1 Notation and Preliminaries 26
3.2 The Suffix Array 27
3.3 SA and LCPA Construction 28
3.4 DivSufSort 32
3.4.1 Additional Notations 32
3.4.2 Basic Operation 32
3.4.3 Inducing the ISA and LCPA, in conjunction with the SA . . 33

4 Dfinder 35
4.1 Block Moves Detector (BMD) 37
4.1.1 **Type 1:** Matching segments found in the current firmware (copied in forward order) 39
4.1.2 **Type 2:** Matching segments found in the current firmware (copied in reverse order) 40
4.1.3 **Type 3:** Matching segments found in the partially reconstructed updated firmware (copied in forward order) 41
4.1.4 **Type 4:** Matching segments found in the partially reconstructed updated firmware (copied in reverse order) 41
4.2 Optimiser 43
4.3 In-place Reconstruction of the Updated Firmware 46
4.4 Orientation of Dfinder to Executable Files 47

5 OTAP Testbed 49
5.1 IoT Technologies and Standards 49
5.1.1 IEEE 802.15.4 50
5.1.2 6LoWPAN 50
5.1.3 RPL 51
5.1.4 Constrained Application Protocol (CoAP) 51
5.1.5 OMA Lightweight Machine to Machine (LwM2M) 52
5.1.6 Eclipse Leshan 53
5.1.7 Contiki-NG 53
5.1.8 RPL Border Router 53
5.1.9 Zolertia RE-Mote Platform 54
5.2 Testbed Implementation 55
5.2.1 OTAP Server 55
5.2.2 OTAP Client 57
5.2.3 Device Preparation 58

5.2.4 Delta Script Download and Firmware Installation 59
5.2.5 Firmware Reconstruction 62

6 Experimental Evaluation **63**
6.1 Evaluation using Randomly Generated Binary Files 64
6.2 Evaluation using Actual Firmware Images 69
6.2.1 Optimised Code 70
6.2.2 Unoptimised Code 72
6.3 Energy Consumption and Dissemination Performance 73

7 Conclusion and Future Work **79**

8 Acknowledgments **81**

Bibliography **83**

List of Tables

2.1 Summary of differencing algorithms 15

3.1 The suffixes of sequence *"banana"* 27
3.2 The SA, ISA and LCPA computed on sequence *"banana"* 28

4.1 Types of matching segments *BMD* is able to detect for each segment of the updated firmware 35

5.1 Manifest structure 57

6.1 The benchmarked differencing algorithms 63
6.2 The different firmware versions that we inputted to the differencing algorithms 69
6.3 The size of the compiled firmware images using the default optimisations 70
6.4 Delta script size (in bytes) produced by the benchmarked differencing algorithms for optimised code 70
6.5 Mean execution time (in seconds) of differencing algorithms for optimised code 71
6.6 Peak memory consumption (in bytes) of differencing algorithms for optimised code 71
6.7 The size of the compiled firmware images without linker optimisations 72
6.8 Delta script size (in bytes) produced by the benchmarked differencing algorithms for unoptimised code 72
6.9 Mean execution time (in seconds) of differencing algorithms for unoptimised code 73
6.10 Peak memory consumption (in bytes) of differencing algorithms for unoptimised code 73
6.11 The predefined energest types 74
6.12 Electric current per operation/state of CC2538 SoC 75
6.13 The delta script size (in bytes) produced by the benchmarked differencing algorithms for an LwM2M-enabled Contiki-NG application 75

List of Figures

1.1 Total number of IoT connected devices worldwide (2019-2030) . . . 1
1.2 Typical scenario of an interconnected sensor network 2
1.3 Common segments between two firmware versions 6

2.1 The steps for wirelessly updating an IoT network, using delta encoding 10

3.1 Median execution time of SACAs versus input file size 31
3.2 Median memory consumption of SACAs versus input file size . . . 31

4.1 *ADD* instructions format 36
4.2 *COPY* instructions format 36
4.3 The main components of *Dfinder* 37
4.4 The construction of byte-array T as a composition of the two firmware versions, the current and the updated one 38
4.5 Finding *Type 1* matching segments 40
4.6 Finding *Type 2* matching segments 40
4.7 Finding *Type 3* matching segments 41
4.8 Finding *Type 4* matching segments 42
4.9 *Copy_type1_rel* and *Copy_type3_rel* instructions format 47
4.10 *Copy_in-place* instructions format 48

5.1 An IoT protocol stack 50
5.2 LwM2M architecture 52
5.3 An interconnected IoT network, using a border router 54
5.4 Zolertia RE-Mote Platform 55
5.5 List of IoT devices registered to the Leshan server 56
5.6 The available resources and methods offered by an LwM2M client, when the device runs the firmware image 59
5.7 The available resources and methods offered by an LwM2M client, when the device runs the firmware installer 60
5.8 Flow of firmware upgrade from v1.0 to v2.0 using delta image . . . 61

6.1 The size of the resulted delta script size versus the unsimilarity (expressed by the Levenshtein distance) of the two input binary files . 66

6.2 Execution time of differencing algorithms versus the unsimilarity (expressed by the Levenshtein distance) of the two input binary files 67
6.3 Execution time of differencing algorithms versus the total size of the two input binary files 67
6.4 Peak memory consumption of differencing algorithms versus the accumulated size of the two input binary files 68
6.5 Time in seconds needed for updating the target IoT device 76
6.6 Energy consumption during firmware update 76
6.7 Energy consumed by the target IoT device per operation/state during firmware update 77

Chapter 1

Introduction

1.1 Internet of Things (IoT)

The ever-expanding evolution of big data and artificial intelligence (AI) has revealed their necessity for building the future smart society (Society 5.0)[1]; a human-centered society that balances economic prosperity with the resolution of social issues by a system that integrates both cyberspace and physical space. In this society, the Internet of Things (IoT) plays a foremost role, connecting a soaring number of islands of smart objects, robotics, and devices, empowered by the accelerated adoption of advanced cellular technologies, such as 5G. The information collected by these smart devices (wearables, electric vehicles, sensors, etc.) can be used for refining the AI algorithms and allow them to make more precise decisions without any human involvement, greatly benefiting both the industry and human

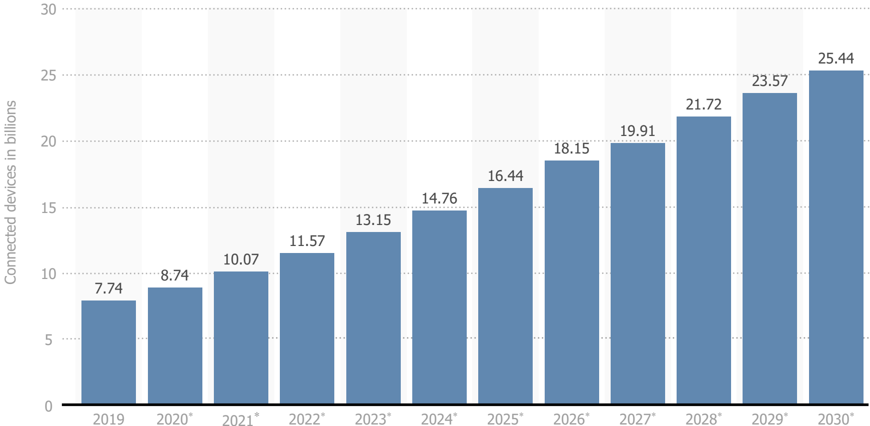


Figure 1.1: Total number of IoT-connected devices worldwide (2019-2030)[2]

[1] https://www8.cao.go.jp/cstp/english/society5_0/index.html

[2] Source: www.statista.com/statistics/1183457/iot-connected-devices-worldwide

life. The significance of IoT can also be showcased by the increasing number of IoT-connected devices, while recent studies predict that by the year 2030 the total number of them will reach 25.44 billion, as illustrated in Figure 1.1. Subsequently, IoT has attracted the interest of academic circles, with them developing new technologies for supporting this multi-protocol and multi-platform infrastructure we refer to as IoT.

1.1.1 Wireless Sensor Networks (WSNs)

Wireless Sensor Networks (WSNs) are Low-Power and Lossy Networks (LLNs) and constitute an early form of ubiquitous information and communication networks, comprising one of the fundamental blocks of IoT. They are self-configuring networks of low-cost, radio-equipped devices (called *motes* or *nodes*), which have constrained resources [25] and are deployed in quantity to monitor or control their surrounding environment. The information gathered by these devices often needs to be uploaded to the cloud, so that it can be processed accordingly. To this end, WSNs also integrate border routers in their topology, which allow the seamless communication of the nodes with the internet, as the limitations that govern such networks do not allow this communication to be performed natively. Hence, the border router acts as a sink, receiving all the messages that originate from the nodes of the network and forwarding them towards the internet, and vice versa. Besides, a WSN may be heterogeneous, in the essence that the devices that it consists of can have different functions/roles, manufacturer, etc. The application areas such networks have been used to, cover a wide spectrum of needs and domains, ranging from smart cities and healthcare to industrial automation and military surveillance, to name a few.

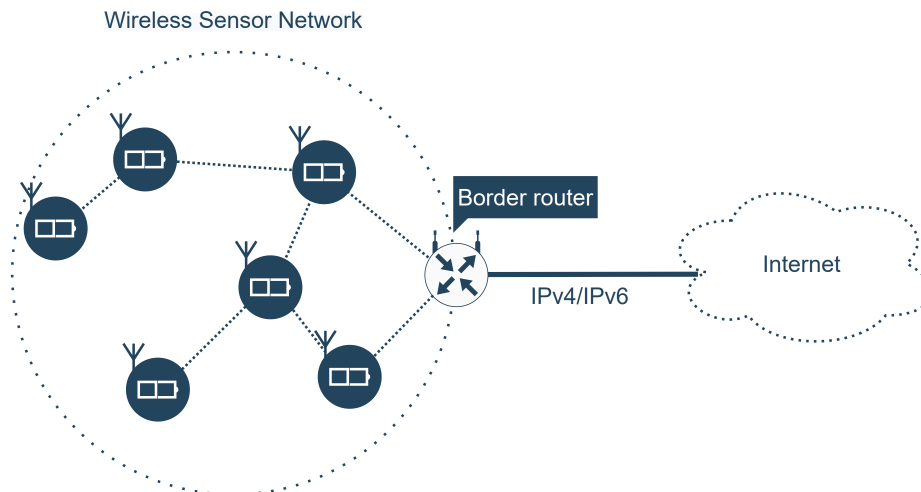


Figure 1.2: Typical scenario of an interconnected sensor network

In general, a node contains a microcontroller, a wireless module, memory, sensors/actuators, as well as a power source (often in the form of a battery), while

some nodes may have different configurations, depending on their purpose in the network. According to their roles, nodes can be classified into *sensors*, *sinks* and *actuators*. Sensors are typically equipped with various kinds of sensing modules and collect information about their surrounding environment, which they transmit to the sink nodes. On the other hand, upon receiving this information, the sink nodes pass it to the border router so that it can reach the internet. Besides, actuators are devices that are used for controlling their surrounding environment (e.g. switching on/off LED lights), based on sensor readings and other user inputs. Additionally, depending on the application and the geography, the nodes may form various network topologies (e.g. ring, star, mesh, etc.). Finally, the software these devices run to serve their purpose is called firmware and is accommodated in their internal flash memory; the flash is also used for running the firmware from, as the RAM is too limited for this task.

1.2 Over-The-Air Programming (OTAP)

Regardless of the attention given during firmware development, software bugs can occur at any level of the system and stage of the development cycle [21], and their effects may be realized after the deployment of the network, degrading the Quality-of-Service (QoS) or even the integrity of it. Hence, firmware updates are frequently released either for resolving bugs [87] or security vulnerabilities [45], or for supporting additional features. On the other hand, the inability to update an IoT network can result in decreased network performance, security breaches that could compromise the privacy and safety of the network, and in general, undermine the long-term sustainability of the deployment. Additionally, the IoT devices found in some deployments may need to be completely repurposed, as they may be assigned different roles, due to changes in their environment. Thus, as their storage is too limited to accommodate multiple application images simultaneously, the new firmware should be compiled at a firmware server and then used to flash these devices. In the most common scenario, however, updates introduce small modifications to the firmware code, from one version to the next, modifying the implementation of a few functions and reconfiguring some application parameters [83].

Traditionally, in order to update a WSN, maintenance personnel had to be dispatched, collect the devices, and access them via a serial port or another hard-wired back-channel, so that they can flash them with the new firmware image. The limitation of this solution, however, is the lack of scalability, and the vast amount of time it requires, which may not be tolerable when an update includes security patches that should be installed promptly. Furthermore, physical access to these devices may be infeasible sometimes, as they may be located in inaccessible areas (e.g. implanted into asphalt roads), or even implanted into human bodies, operating as medical sensors [97].

To alleviate the above limitations, in the early 2000s were developed the first systems that offered Over-The-Air programming (OTAP) support for WSNs [90].

These systems included a firmware server that compiled the updated firmware version and then transmitted it to its neighboring devices, over a radio channel. Once a device had fully received the update, it rebooted, the bootloader overwrote the contents of the internal flash memory (program memory) with the new firmware and started running the updated version[3]. A limitation of these preliminary solutions, however, was that only the devices that were within the radio range of the server could receive the update. Nowadays, due to the large geographical scale of modern IoT networks, multi-hop communication is a reality, and to facilitate proper OTAP, more advanced dissemination protocols have been designed, whose goal is the reduction of the energy consumption in the network, avoiding redundant transmissions and collisions that can degrade the quality of the channel.

1.3 Problem Statement

When designing OTAP solutions, the limitations that bind the LLNs [57], due to their constrained nature, should be taken into account. For instance, IoT devices are mostly battery-powered and operate unattended in harsh environments for extended periods of time [24], while some of these devices may rely on ambient power sources such as solar energy [27], wireless energy, and RF. A node's lifespan is greatly affected by routine operations such as those involved in wireless radio communication. For instance, the transmission of a single bit of data could consume roughly the same amount of energy as executing 1000 instructions [78]. Writing data into the flash memory is not a lightweight process either, as a higher voltage is required, resulting in additional power consumption. All the above conclude that the required time for updating a network, as well as the total power consumption, are significantly affected by the size of the data that need to be transmitted to the network and stored by the target devices, during a network upgrade.

However, the microcontrollers found in motes are not suitable for running complex cryptographic (RSA [69], AES [20]), or compression algorithms, as the execution time will be significantly high, during which the node may be unable to perform any other operations. Regardless, the energy consumption caused by these operations would render them unsuitable for such energy-constrained environments. To address this limitation, the research community works towards lightweight algorithms [65], and hardware-accelerated cryptography operations [8].

Additionally, it is very common for IoT devices to have a limited NAND-based flash memory [79], which needs to accommodate both the bootloader and the firmware code. The situation gets harsher if we consider that a portion of the flash memory may be occupied for storing rollback images (golden images), for providing fail-safe OTA updates. Hence, a firmware update strategy should optimally utilize the available flash memory and give great emphasis on reducing its degradation, caused by excessive erases (e.g. storing temporal data in the RAM [39, 18, 51, 86]).

[3] Throughout this work, we refer to the old and new firmware versions as the current and updated one, respectively.

Based on the above, a solution is needed for reducing the transmitted data during a network upgrade, without applying a traditional compression algorithm. Besides, this solution has to make very efficient use of the limited flash memory included in IoT devices.

1.4 Incremental Programming

As developers used to release firmware updates regularly, transmitting the entire firmware when a new version had been released, quickly became obsolete, triggering the development of the first *incremental programming* schemes [68, 3]. In the context of firmware updates, incremental programming is a technique that aims at reducing the volume of transmitted data during a network update, by avoiding the transmission of the entire firmware. It is based on the fact that two versions, of the same firmware, share many structural similarities, and hence, some parts of the new version may not need to be transmitted, as the receivers already possess them as parts of the version they currently run. In the literature, incremental programming has been introduced in various forms, some of which will be discussed in the next chapter; however, in this study, we will focus on the one that is based on delta encoding.

As the server knows the firmware version that the nodes currently run, it could compute and transmit an encoded patch, which describes the differences between the two firmware versions: the firmware version that the server just compiled and the one that the nodes currently run. Upon receiving this patch, the nodes can apply it on the current version to reconstruct the updated one, locally ($f_{Current} \circledast f_{Patch} = f_{Update}$). Hence, instead of transmitting the updated firmware, the server could simply disseminate this patch, which is significantly smaller than the firmware image itself, reducing the energy consumption and the time needed for updating a network.

The process of computing such a patch of minimal size between two firmware versions is often referred to as delta encoding or differential compression [88] and is done by a *differencing algorithm*. This patch, or delta script (these terms will be used interchangeably in this study), formally describes the differences (non-common segments) between two file versions, and also contains a set of instructions that the nodes need to perform, so that they reconstruct the updated one, using parts of the version they currently have. For instance, as shown in Figure 1.3, only A', D' and F' need to be explicitly transmitted, as they are not common between the two firmware versions, while the remaining segments could be reconstructed by the receiver, simply copying the corresponding (matching) segments from the current firmware image.

Differencing algorithms can be further subdivided into block-level (e.g. [37, 93]) and byte-level (e.g. [19, 32]) algorithms, according to the level of granularity they operate at, to detect common segments between two files. The byte-level algorithms usually produce smaller delta scripts, as they are more flexible and can

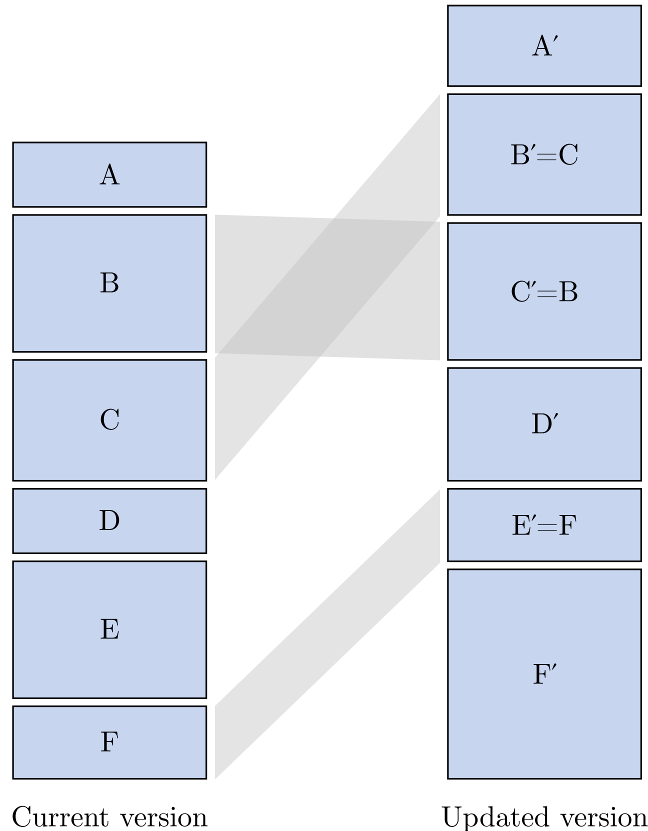


Figure 1.3: Common segments between two firmware versions

detect more common segments with varying lengths, while the block-level counterparts, typically have lower execution time and memory footprint but produce larger delta scripts. The performance of a differencing algorithm is determined by three factors [33]: (i) the compression ratio, (ii) the time complexity, and (iii) the space complexity. Unfortunately, these three indices constraint each other, as often an algorithm that can produce small delta scripts suffers from poor execution time and memory footprint.

There has been an effort to create universal delta script formats (e.g. *VCDIFF* [47], *GDIFF* [4]), while some algorithms use custom ones [19, 17] that support additional instructions in order to achieve a more efficient encoding. Despite their differences, all proposed formats feature two core instructions, with a relatively standardized syntax: *COPY* and *ADD*. A *COPY* instruction is used for encoding a segment of the updated firmware that is common between the two firmware versions and subsequently does not need to be transmitted. On the other hand, an *ADD* instruction encodes a non-common segment of bytes, found in the updated firmware, which needs to be transmitted to the target nodes. Besides, some formats, such as *VCDIFF*, include an additional instruction, called *RUN*, which is

used for encoding bytes of the updated firmware which are repeated multiple times (e.g. segments that consist of many zeros). It is evident that in order to produce smaller delta scripts, the differencing algorithms should aim at reducing the number of *ADD* instructions, as the respective non-common byte segments contribute mostly to the delta script size. Thus, they should work towards maximizing the number of common segments found between two versions, so that they can produce smaller delta scripts.

1.5 Contributions

In this thesis, we present a byte-level differencing algorithm we have developed in ANSI C, called *Dfinder*, that uses *enhanced suffix arrays* [1] and can generate small delta scripts, very efficiently (both in time and memory). *Dfinder* can detect various types of common segments between two firmware versions, effectively increasing the ratio of the bytes of the updated firmware that can potentially be encoded using *COPY* instructions. Namely, for a segment of the updated firmware, *Dfinder* can find a matching one in the current firmware, as long as the latter can be copied in forward or reverse order to reconstruct the former segment. For instance, both *"abcde"* and *"edcba"*, found in the current version, could be copied to reconstruct the segment *"abcde"*, found in the updated one. Besides, the algorithm can utilise parts of the updated firmware that have already been reconstructed to find matching segments for following segments (of the updated firmware). For example, if the updated version was *"abcdeabcde"*, the first half of it could be copied to reconstruct the second half. Additionally, the time and space complexity of the algorithm are kept low ($O(nlogn)$ and $O(n)$) and takes less than 0.05 seconds to compute a delta script for all Contiki-NG[4] applications (up to 80 KiB) that we tested for Zolertia RE-Mote platform[5].

Furthermore, a drawback of traditional *incremental programming* approaches is that the nodes are required to have enough storage space to accommodate both firmware versions, simultaneously, during the reconstruction process. However, this may not be feasible in some deployments, due to cost limitations. To alleviate this challenge, we propose an extension of *Dfinder* that allows the receivers to perform the reconstruction of the updated firmware in-place, starting at the flash address where the current firmware version is stored. Subsequently, in order to be updated incrementally, IoT devices need to have no extra space than that needed for storing the base firmware version, in the first place. Furthermore, we integrate *Dfinder* in an OTAP testbed that we implemented, and evaluate its performance, comparing it with other differencing algorithms. Besides, using the testbed we showcase how the time and energy needed for updating IoT networks are affected when an incremental approach is followed.

[4] `www.contiki-ng.org`
[5] `www.zolertia.io/product/re-mote`

Chapter 2

Related Literature

The limitations imposed by the inherent restricted nature of IoT networks have forced the academia and the industry to develop OTAP strategies, focused on reducing the transmitted data volume during network upgrades. Some of these approaches constitute a form of incremental programming, as their operation is based on reusing parts of the previous version for the reconstruction of the updated one (e.g. proposed differencing algorithms). Besides, different network protocols have been proposed in the literature, that enable the efficient and robust dissemination of the update, while some of them are focused on securing the communication by applying cryptography algorithms. In this chapter, we present some of these contributions and we compare them to our work.

Virtual machines partially overcome the challenge of large updates, by running interpreted code. Since byte code is typically much smaller than the native counterparts, updates in such systems are also smaller and easier to be distributed. The idea of VMs running on resource-constrained devices was first implemented in *Maté* [55] and *VM** [49]. Although VMs seem a promising way for reducing the update size, they also introduce latency, as the interpreted execution is generally slower, while some resources are constantly occupied by the VM itself. Besides, this approach does not work for updating the core components of the virtual machine (engine), as in this case, the whole updated VM image has to be transmitted to the target devices.

The main advantage of modular operating systems over non-modular ones is the support of dynamic linking [21] and loading. This way, systems such as Contiki OS [22] [1], only need to receive the modified modules of the updated firmware image, as the dynamic linker can use them for re-linking the image and loading it again. However, apart from the modified module(s), the new symbol and relocation tables also need to be transmitted, contributing to the transmission volume significantly. *Elon* [18] avoids the transmission of the relocation table, introducing the concept of replaceable components. A drawback, however, is that these

[1] Dyncamic linking/loading was supported by the preliminary version of Contiki-OS, but is not supported in Contiki-NG.

components are stored in the RAM of the devices, and subsequently need to be re-transmitted in case of a reset, which is common for WSN nodes. Regardless, an OTAP strategy based on the support of dynamic linking at the receiver implements the incremental programming paradigm, because the unmodified modules do not need to be transmitted, as the receiver already possesses them and can use them for the construction of the updated image.

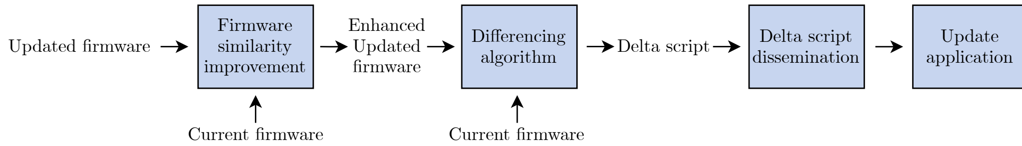


Figure 2.1: The steps for wirelessly updating an IoT network, using delta encoding

Incremental programming, based on delta encoding, has attracted the attention of the research community, presenting as a promising approach for the reduction of the transmitted data, during a network upgrade; this is the technique we will focus on in this work and mainly discuss in this chapter. As illustrated in Figure 2.1, the main steps for updating an IoT network using delta encoding are as follows.

- **Firmware similarity improvement**, which includes the utilization of techniques that mitigate the effects of function and variable shifts in the updated firmware image, and increase the similarity between two firmware versions; hence, potentially resulting in the reduction of the computed delta script size.
- **Differencing algorithm application**, including the utilisation of a differencing algorithm on two firmware images, in order to compute a delta script. The delta script contains a set of instructions that, once applied to the current firmware, enable the reconstruction of the updated version.
- **Delta script dissemination**, which is responsible for orchestrating the efficient and reliable distribution of the delta script to the devices of the network, applying a suitable network protocol that focuses on avoiding excessive transmissions.
- **Update application**, which refers to the OTAP stage that takes place at the node, once it has fully received the delta script, and includes the reconstruction, verification, installation, and execution of the updated firmware. This step is strictly connected to the patch generation process and the chosen delta format, so that the delta script can be interpreted accordingly.

2.1 Improving Firmware Similarity

Although firmware updates usually introduce minor modifications to the firmware source code, these changes can result in the disproportional increase of the computed delta script. In [82], the authors distinguish four firmware properties that

affect the size of the generated delta script.

Function shifts
In the updated firmware version one or more functions may be modified, altering the size of their implementations. This shrinkage or increase of them can force functions whose implementation is located at higher addresses, to be shifted (to other addresses), to comply with the new size of the modified functions. Subsequently, calls towards these shifted functions will have different target addresses [2] between the two versions. Finally, as all these altered target addresses are encoded using distinct *ADD* instructions, the size of the computed delta script can be significantly increased.

Global variable shifts
The insertion or deletion of global variables in the source code is another factor that can affect the size of the computed delta script. The global variables of a program are stored in two adjacent sections of the RAM: *.data* and *.bss*. The initialized global variables are mapped to the former section, while the uninitialized ones are mapped to the latter. When a global variable is introduced or removed, the variables that have been mapped to higher addresses are shifted. Hence, similar to function shifts, any reference to these variables will have a different target address between two firmware versions, increasing the delta script size, due to the insertion of additional *ADDs*.

Relative jumps
The jump instructions transfer the program sequence to an absolute flash address, given in the instruction format. Hence the next instruction to be executed is the one present at that flash memory. However, instead of using absolute target addresses, relative jumps are performed based on an offset, that represents the distance between the jump instruction and the target address. The insertion or deletion of instructions, between a relative jump instruction and its target address, will cause the alteration of the respective offset, forcing the injection of additional *ADDs* in the final delta script.

Indirect addressing
In RISC architectures, memory locations can only be accessed indirectly, through the processor registers. Indirect addressing provides fast access to large data structures, such as arrays, linked lists, unions, etc. As global variables are shifted, resulting in the modification of the global variable layout, the corresponding indirect instructions will contain different target addresses (between the two versions), which must be encoded using *ADDs*.

Based on all the above, it is paramount to preserve the similarity of two firmware

[2] The flash address where the implementation of the called function originates at.

versions, before the delta script computation, in order to ensure that the resulted delta script will have the smallest size possible. This can be achieved, by mitigating the effects of the shifts using some similarity preserving techniques, such as the ones that are discussed in this section.

2.1.1 Slop Regions

Slop regions were introduced in [48] and since then, they have been adopted in many OTAP schemes for addressing the function and variable shifts. A slop region is defined as a free memory space, located directly after a function's implementation, where the function can grow or shrink without forcing any other functions to relocate. If a function grows, part of its slop region will be utilized for accommodating the new code. On the other hand, if a function shrinks, its slop region will grow, occupying the removed part of the function. Slop regions have also been used between the *.data* and *.bss* sections in RAM, so that the uninitialized variables do not need to be shifted when initialized global variables are introduced or removed in the new version[71].

To mitigate the effects of function shifts, an OTAP scheme, called *Qdiff* [82] proposes an alternative implementation of slop regions. *Qdiff* does not create a slop region for each function during linking, as other implementations do. In contrast, when a function is deleted or shrunk, the resulted available space becomes slop region. Hence, slop regions can be found only immediately after a function's implementation, caused by the shrinkage of its implementation, or by the deletion of other functions. 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 updated function will be relocated at the end of the firmware code. This way, by removing (or shrinking) and/or creating (or expanding) some functions in the updated version, *Qdiff* proactively creates empty slop regions that can be assigned to functions, later on.

In *Qdiff* [82], the authors also follow an interesting approach for addressing global variable shifts. As mentioned above, the *.data* and *.bss* sections are located in adjacent regions in RAM and expand towards the same direction (towards higher addresses). The authors modified the method that is responsible for the expansion of these two sections, so that they expand towards the opposite direction. The two sections originate at fixed addresses, having a large empty space between them and expand towards the opposite direction, progressively filling the space between them. This, way the uninitialized variables are not shifted when initialized ones are removed or inserted in the firmware source code.

A drawback of this technique, however, is that excessive fragmentation of the flash memory may occur, as some regions of the flash may contain actual code, while others being empty slop regions. Apart from the inefficient utilization of the flash memory, fragmentation can also increase energy consumption because the control circuitry needs to activate a larger number of memory regions during the operation to run the firmware image. The authors in [38] show that the energy

consumption can increase by up to 5% when memory is fragmented. Finally, extra attention is needed when a function grows beyond its slop region, as it may need to relocate to a different address.

2.1.2 Position-Independent Code (PIC)

Position-independent code is a piece of software that is allowed to execute regardless of the absolute memory address it is loaded at. If a code is *position-independent*, all references and target addresses in instructions are relative, representing the distance between the calling instruction and the respective destination. Hence, if this piece of code needs to relocate in the updated firmware version, these relative addresses will not be affected, which would not be the case if absolute addresses had been used instead. Subsequently, *PIC* has been used, for improving the similarity of firmware versions before the delta script computation (e.g. in SOS operating system [30]).

However, due to hardware limitations of embedded devices, the offset of the relative jumps can be performed only within a certain range, limiting the application of this technique. For example, Atmel AVR platform supports PIC but restricts the size of the program to 4 KB. Additionally, PIC needs compiler support, which is not the case for some platforms (e.g. MSP platform [21]).

2.1.3 Indirection Tables

The use of *indirection tables* was introduced in *Hermes* [71] and *Zephyr* [72], as a countermeasure against the effects of function shifts. When the updated firmware image is linked, an indirection table is created, that has an entry for each function of the firmware, which is called at least once. Each such entry contains the flash address, where the respective function is located at. This way, all calls toward functions can be replaced by jumps towards the corresponding entries of the table, which will indirectly give control to the called functions.

The advantage of this technique is that when a function relocates, only its entry in the indirection table is affected and needs to be updated, while the target address of all instructions that call this function will stay intact, containing the respective entry of the indirection table.

However, this technique is platform-specific and linker modification is required, which may affect the performance of the code. Moreover, as function calls are performed indirectly, through the table, the run-time latency of function calls increases. Finally, apart from the delta script, the table also needs to be transmitted and stored at a fixed address of the device flash memory. This can be problematic when we use this approach on complex programs with many functions and function calls, as the table size will be significantly increased, resulting in transmission overhead, while a large amount of flash memory may be occupied for the table itself.

2.1.4 Relocatable Code

The *relocatable code* is a software module whose initial address can be dynamically moved around the available address space and loaded in multiple addresses. When such a code is compiled, a relocation table is also created, containing an entry for each reference in the code that should be dynamically resolved, when the program is loaded. Subsequently, it is a common technique for traditional dynamic linking and loading.

In *R2* [17], the authors utilize the concept of *relocatable code* for mitigating the effects of function and variable shifts. Initially, the linker is instructed to to generate relocatable code for the updated firmware image. Then the image is parsed and all references to symbols (functions or global variables), are set to a predetermined value which remains the same across all firmware versions. Each such altered reference has an entry in the relocation table, which contains the correct value. As the references between the two versions will be the same, set to the predetermined value, the effects of the function and variable shifts between the two versions will be mitigated. Then, the relocation table and the altered firmware image are merged and used for delta generation. Upon receiving the delta, the device is able to reconstruct the firmware image and the relocation table, while it can also resolve the addresses using the latter.

A disadvantage of *relocatable code*, however, is that it requires a modular OS at the recipient side, with a sophisticated loader that can resolve the relocated addresses before the firmware executes. It must also be mentioned that although the use of relocatable code and indirection tables seem similar, these techniques have some major differences. The relocatable code must be resolved during load-time, while the indirection table-based code operates at run-time, introducing additional latency. Furthermore, the metadata of the relocatable code are generally larger in terms of size compared to those of the indirection table-based code.

2.2 Differencing Algorithms

In order to achieve more efficient utilization of the available resources in resource-constrained IoT networks during a network update, the transmission volume has to be minimized. This can be accomplished through a process called *delta script generation*, which relies on two observations: (i) both the server and target IoT nodes have the current firmware version, and (ii) the updates mostly introduce small modifications in the firmware code. Thus, only the parts of the updated firmware that are not common between the two versions, need to be transmitted, followed by a set of commands that instruct the node how to reconstruct the updated version, locally. Some of the most popular differencing algorithms are shown in Table 2.1 and discussed in this section.

Algorithm	Granularity	Time complexity	Space complexity
FBC [37]	block-level	$O(n)$	$O(n)$
Rsync [93]	block-level	$O(n^2)$	$O(n)$
Xdelta [58]	block-level	$O(n)$	$O(n)$
BSDiff [19]	byte-level	$O(nlogn)$	$O(n)$
RMTD [32]	byte-level	$O(n^3)$	$O(n^2)$
DASA [63]	byte-level	$O(nlogn)$	$O(n)$
R3diff [19]	byte-level	$O(n^3)$	$O(n)$
DG [40]	byte-level	$O(n^2)$	$O(n)$

Table 2.1: Summary of differencing algorithms

2.2.1 Fixed Block Comparison (FBC)

FBC [37] is the simplest known method for comparing two firmware images, aiming at reducing the required transmissions during a network upgrade. This algorithm divides the two images into blocks of fixed size and compares the respective blocks. For each pair of matching blocks, a *COPY* instruction is inserted into the delta script, while the non-matching ones need to be explicitly transmitted, along with the delta script. In order to encode the latter blocks, an *ADD* instruction has 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, despite its simplicity, it works quite well for small firmware changes, as 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, that can span across many blocks.

2.2.2 Rsync

Rsync [93] (and the corresponding *Rdiff* utility) is a file synchronization tool, that is used by many incremental reprogramming schemes [36, 72], for detecting common segments between two file versions. It operates at block-level and was initially developed for exchanging binary files over low-bandwidth channels. *Rsync* segments the two files into fixed-sized blocks and uses a sliding window with a size equal to the block size, to scan the two files for matching segments.

First, both the rolling-checksum and a MD4 checksum of each block of the old version are calculated. Then, the window traverses the new file in a sliding window fashion, and the digest of the current window gets computed. This way, potential matches for the window can be detected, conducting a lookup with the digests of the old file blocks. Similar to other sliding window protocols, if a match is found for the current window, the window moves forward one block, otherwise

it moves one byte, signing the first byte of the current window as unmatched. All these unmatched bytes are accumulated and transmitted when the next matching is found, or when the window reaches the end of the old file.

Although *Rsync* can find common subsequences with higher accuracy compared to *FBC*, it still faces the same drawbacks, since its granularity depends on the used window size; thus, being unable to detect common segments with a size smaller than that of the used window. For instance, if two respective blocks of two file versions differ by a single byte, the entire block of the new file has to be explicitly transmitted. Additionally, being a general-purpose differencing algorithm, *Rsync* does not use any file-specific knowledge and is outperformed in the context of executable files, by more specific tools, such as *BSDiff* and various versions of *Xdelta*. Moreover, *Rsync* cannot update files in place, as the output file must not be the same as the input one.

2.2.3 Xdelta

Xdelta [58] is a linear time and space differencing algorithm that operates at block-level and generates delta scripts in *VCDIFF* format. The algorithm merges the two input files, constructing a new file, which is compressed using LZ77 [89] (or another compression algorithm), compressing only the part that contains the new file. As the two file versions share many common segments, the updated version will be efficiently compressed. In this essence, data compression can be considered as a special case of differencing, where no old file is provided. Besides, *Xdelta* results in small delta size by optimizing the generated instruction set and merging small instructions into one; the generated delta script consists of a sequence of instructions (ADD, COPY, or RUN).

Additionally, a trait of this application is that it prioritizes speed over compression performance and is favored when the delta computation time is the most important factor. However, in normal operation mode, *Xdelta* consumes a significant amount of memory, which may be intolerable if the firmware server computes multiple patches, simultaneously. Besides, the nodes have to be instructed to interpret the *VCDIFF* format, so that they can reconstruct the updated firmware, properly.

2.2.4 BSDiff

BSDiff [73] is a differencing algorithm that focuses on executable files, and uses suffix sorting for the efficient computation of patches. First, the two files are scanned both forwards and backwards, so that segments of the new file, that exactly match with others of the old file, are detected. Next, *BSDiff* computes approximate matches, expanding the detected matching segments in either direction, so that every suffix/prefix of the extension matches in at least half of its bytes. These last matches correspond to slightly modified segments of the updated firmware, that have small differences between the two versions. For instance, these can be regions

that contain calls to shifted functions. Additionally, the main parts of the resulted delta script (instructions and matches) are further compressed using the bzip2 compression library [3]. Besides, the algorithm relies on common change patterns observed in executable code, so that it produces smaller patches for executable files, compared to other such tools. Finally, *BSDiff* has $O((n + m) * logn)$ time complexity and requires no more than $max(17*n, 9*n+m)+O(1)$ bytes of memory, for its execution.

2.2.5 RMTD

RMTD is a byte-level differencing algorithm proposed by Jingtong Hu et. al. in [32]. A novelty of this algorithm is that in order to encode a segment S of the updated firmware, it can also detect matching segments in the part of the updated firmware that will have already been reconstructed when S is about to be reconstructed. As the instructions in the delta script are sequentially executed by the recipient, the updated version is progressively reconstructed. Thus, when a segment of the updated firmware, encoded in an instruction (*ADD* or *COPY*), is about to be reconstructed, the part of the updated firmware that precedes it in order, will have already been reconstructed in flash memory. In addition, *RMTD* can detect common segments between two firmware versions, in both forward and reverse order. For instance, the segment *"edcba"* is considered as a matching one for the segment *"abcde"*, and the one could be used to reconstruct the other.

RMTD uses a large 2D matrix for recording the common bytes of the two firmware images. Based on this matrix, the algorithm can detect all the pairs of matching segments. 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 updated image and the rest of the (updated) image, respectively. Once these two lists are computed, the algorithm finds the optimal combination of *COPY* and *ADD* instructions to encode the segments of the updated firmware, using a dynamic programming approach.

Being able to utilize the partially reconstructed part of the updated firmware and detecting segments that can be copied in reverse order to reconstruct their matching ones (in the updated firmware) enables *RMTD* to generate very small delta scripts. However, the algorithm has been criticized for its excessive time and space complexity ($O(n^3)$ and $O(n^2)$) that render it unusable for modern large-scale applications [63]. The authors in [63] have reported that *RMTD* crashed when the code size becomes too large ($\sim$ 42 KiB), due to lack of memory resources.

2.2.6 DASA

DASA [63] is a differencing algorithm proposed by Biyuan Mo et. al., aiming at producing delta scripts, keeping the space and time complexity low. In order to accomplish this, the algorithm utilizes an efficient data structure, called *suffix*

[3] `www.sourceware.org/bzip2`

array (*SA*) [15]. First, *DASA* constructs a byte-array, as the composition of the two firmware images, which is used to compute the *SA* using the *doubling algorithm* [15]. Precomputing the *SA* as well as some auxiliary arrays, the algorithm can conduct efficient queries, that allow the detection of common segments (in forward order) between two firmware versions.

The authors implemented a function called *findK*, that given an index i of the updated image, returns the smallest possible index k, so that the subsequence of the updated firmware that originates at the kth byte and ends at the $(i-1)^{th}$ one is a common segment (found both in the updated and current image), and thus could be reconstructed copying the respective segment from the current firmware. Besides, to compute the optimal delta script size, the authors follow a dynamic programming approach.

In experimental evaluation [63] *DASA* outperformed *Rsync* in terms of delta script size, which is expected, as *Rsync* is a block-level algorithm. Furthermore, *DASA* has better execution time and smaller memory footprint than *RMTD*, which is heightened when the firmware image is large.

2.2.7 R3diff

R3diff [19] is a byte-level differencing algorithm, which is heavily inspired by its predecessor; *DASA*. *R3diff* can only detect common segments between the two firmware versions in forward order, while it also adopts a dynamic programming approach, similar to the one presented in *DASA*, for the computation of delta scripts. However, instead of using *SAs*, the algorithm computes the hash value for every three bytes of the current image, located in consecutive order and the *findK* method operates according to these digests.

2.2.8 Delta Generator (DG)

The authors in [40] have presented another differencing algorithm, called *Delta Generator* (*DG*). The algorithm places the two firmware images side-by-side and performs a XOR operation between the respective bytes, aiming at revealing the non-matching sequences of bytes, as the matching ones can easily be encoded using *COPY* instructions. Then, these non-matching sequences of the updated firmware are used for detecting the common and non-common subsequences between the two versions.

The algorithm has $O(n+m)$ space and $O(nm)$ time complexity, where n is the size of the current firmware image and m the size of the non-matching sequences found after XOR, and subsequently, the execution time and the memory footprint heavily depends on the structural similarity of the two versions. A comparative study of *DG* and *R3diff* was conducted in [54], where the authors inferred that *DG* outputs significantly smaller delta scripts than *R3diff* for small-sized images. However, this statement does not hold, as more data and code is shifted. Finally, the authors conclude that *DG* is not able to provide optimization for a high number

of small changes; instead, it generates several *ADD* instructions that encode regions with a few bytes for each non-matching segment.

2.3 Dissemination Protocols

The protocols designed for distributing data in WSNs are not suitable for disseminating firmware updates for several reasons. First, the update size is typically much larger (in the order of kilobytes) than that of the data that are usually transmitted, while the protocols have not been optimised for such large payloads. Second, while the flow of the data transmissions in a WSN is bidirectional, including operation information from sensors and commands toward actuators, etc., the flow of the update image is mostly one-way, from the firmware server to the nodes.

Hence, purpose-specific network protocols have been designed, that focus on the efficient dissemination of firmware updates from a firmware server to the nodes of an IoT network. These protocols have to cope with the unreliable nature of the wireless medium, employing mechanisms for the provision of a reliable firmware update process. Regarding this, the nodes should be able to provide feedback, in the form of positive or negative acknowledgments (ACKs), indicating the correct reception of packets, or request the retransmission of lost ones. Additionally, it is evident that flooding the network with the packets is not a viable solution, as it will result in redundant transmissions, depleting the nodes' battery, while it can also cause *broadcast storm problem* [94], where overlapping radio signals result in increased contention and packet collisions. Hence, these protocols should be designed to reduce the number of needed transmissions, allowing more efficient dissemination. so that they comply with the restricted nature of these networks. Besides, some network protocols designed for this task [13, 29, 16] use *pipelining* to rush the upgrade of IoT networks. When pipelining is used, a node upon receiving some chunks of the update can distribute them to its neighbors, acting as a source for them. An extensive survey on the update dissemination protocols is presented in [11].

Besides, the injection of faulty firmware images and their installation by the nodes can manipulate their behavior. To alleviate this risk, some dissemination protocols focus on securing the process, by ensuring the authenticity and integrity of the received firmware image. Again, the support of security has to be implemented efficiently, which will not deplete the battery of the nodes. Digitally signing the whole firmware image and requiring each node to verify it, once it has received it, would increase the time needed for upgrading a network significantly. The reason is that each device should first receive the firmware image in its entirety, verify it and then propagate it to its neighbors, preventing the pipelined dissemination of the update in the network. In addition, if an error occurs during the dissemination, the node will be able to detect it only once it has received the whole update. A more efficient, and secure approach is to divide the update into pages and include the digest of each page in the payload of the previous one. This way, a node can verify

each page upon reception, enabling the pipelined dissemination of the update in a secured way.

2.3.1 XNP

XNP [90] is the earliest network protocol that focused on the upgrade of resource-constrained networks. The protocol transmitted the entire firmware image and was able to reach only the nodes that were only one hop away from the firmware server. The firmware server divides the update into packets and broadcasts them sequentially. Once a node has fully received the update, the bootloader copies the code to the flash memory and restarts the node [96].

2.3.2 Trickle

Trickle [56] is an update dissemination protocol, originally built for the Mica-2 motes [4], that follows a "polite gossip" approach for propagating the updated firmware image throughout the nodes of a network. In *Trickle*, each node periodically broadcasts an announcement that contains the current firmware version it runs, informing others about potential updates.

The update process originates when a node overhears that another node can provide a newer firmware version, or if it receives a broadcast by another one, indicating that it executes an older version. This "polite gossip" approach renders *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(s).

2.3.3 Deluge

Deluge [13] was built for the Mote-2 devices as the default network reprogramming protocol of TinyOS. *Deluge* employs a negotiation mechanism based on *Trickle* and uses pipelined firmware dissemination for increased performance.

Using *Trickle*, 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. *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

[4] `www.cmt-gmbh.de/Mica2.pdf`

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.

A disadvantage of *Deluge*, however, is that it requires the nodes' radio to be constantly turned on, which results in increased energy consumption. Moreover, the protocol does not support fault detection or recovery mechanisms. Finally, the authors have proposed some optimizations on the protocol based on forward error correction (FEC), using the *digital fountain approach* [12], a method used for efficient transmission of bulk data by heterogeneous nodes.

2.3.4 Multihop Network Reprogramming (MNP)

MNP [51] is a multi-hop reprogramming protocol for TinyOS, designed for the Mica-2 and XSM [75] nodes. The protocol aims at reducing the network collisions that occur during the firmware dissemination by proposing a source node selection heuristic, which guarantees that at any given time, at most one node can act as a firmware source for a given neighborhood.

The nodes in a neighborhood broadcast packets that contain the ID of the node that they prefer to act as a source. The one with the highest popularity starts transmitting, while the other potential sources suppress themselves. Once it has finished, it falls in an idle state, allowing other nodes to be selected as sources. Besides, *MNP* improves propagation performance by supporting pipelined dissemination.

2.3.5 Seluge

Seluge [34] is a secure extension of *Deluge*, that provides integrity assurance for the received firmware and also provides resistance against DoS attacks that specifically target the firmware dissemination process. *Seluge* provides direct authentication of each received packet, defeating the DoS attacks that exploit the authentication delays the nodes face by waiting to receive the rest of each 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; however, it still takes a vast amount of time for an attacker to forge the authenticator.

The firmware image is divided into fixed-size pages and each page is further subdivided into packets and for each one of them, the respective digest is computed. Starting from the last page of the updated firmware, the digest of each packet will be appended to the payload of the corresponding packet of the previous page. Namely, the packets of each page will contain the digests of the packets of the next page. This is an iterative process, forming a hash tree [84] that is used as the basis for the computation of the final signature.

2.3.6 Sluice

Sluice [53] is based on *Deluge* and uses *hash chains* for ensuring the authenticity and integrity of the received firmware image, and it also provides pipelined dissemination of the update. Similar to *Deluge*, *Sluice* divides each firmware image into several pages and integrates signatures and hash functions for efficient code authentication. More specifically, the digest of each page is computed and is appended to the payload of the previous page, forming a chain of hashes (hash-chain). Following the concept of digital stream signing, only the head of the hash-chain (first page of the chain) is signed using the private key of the firmware provider, requiring only one signature to be computed for the whole update. For this digital signature, the *ECDSA* algorithm is used with 160-bit SHA-1 hashes.

Using this digital signature, a node can verify the source of the update and can also ensure the integrity of the image, comparing the digest of each page with the one stored in the payload of the previous one. Thus, there is minimal overhead for the nodes, as the only operations required are a single signature validation and the computation of the hash value for each received page.

2.3.7 Secure firmware updates using open standards

The authors in [99] propose a firmware update mechanism based on open standards such as *CoAP*, *LwM2M*, *SUIT*, etc. 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 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.

2.3.8 ASSURED

The authors in [5] propose a scalable architecture for OTAP, supporting end-to-end security. They distinguish four types of stakeholders: (i) original equipment manufacturer (OEM), (ii) firmware distributor, (iii) domain controller, and (iv) 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. 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: (i) on *Hydra* [23], 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 (ii) on ARM Cortex-M23 MCU that is equipped with Trustzone security

extensions [74] and is able to partition the system into two regions (secure, non-secure).

2.4 Comparison of the Related Literature and our Contribution

As noted in the previous chapter, *Dfinder* is a differencing algorithm that enables the efficient computation of small delta scripts. With that in mind, it is useful to examine how it compares to other such algorithms, regarding the compression ratio, as well as the execution time and the memory footprint. First, to the best of our knowledge, apart from *Dfinder* only *DASA* and *BSDiff* utilize suffix sorting for the computation of delta scripts, resulting in remarkably low space and time complexity. Additionally, it is a byte-level differencing algorithm, resulting in smaller delta scripts than the block-level counterparts (e.g. *FBC*, *Rsync*, *Xdelta*).

However, comparing *Dfinder* with some academic contributions, such as *DASA* and *R3diff*, our algorithm can encode a larger ratio of bytes (in the updated version) using *COPY* instructions, as it can utilise the partially reconstructed part of the updated firmware for detecting matching segments, and these matching segments can also be copied either in forward or reverse order to reconstruct the respective segments of the updated firmware. This trait allows *Dfinder* to compute smaller delta scripts, especially when the modifications are extended. These are the matching segment types that *RMTD* can also detect, with significantly higher time and space complexity. Furthermore, compared to *BSDiff*, *Dfinder* does not apply compression on any part of the delta script, resulting in less processing overhead for the nodes.

Besides, *Dfinder* enables the in-place reconstruction of the updated firmware, originating at the same flash address, where the current fimware version is stored, so that nodes with limited flash storage can also be updated, incrementally. Finally, instead of mitigating the effects of function and variable shifts in the updated image, *Dfinder* follows a platform-agnostic approach and processes the resulted delta script itself, reducing its size. This way, *Dfinder* does not require a priori knowledge about the firmware structure and does not depend on sophisticated OSes, allowing its seamless port to other platforms and OSes.

Chapter 3

Enhanced Suffix Arrays and Applications

The generic name *enhanced suffix array* stands for the set of data structures that include the *suffix array* (*SA*), as well as some auxiliary arrays (*LCP array*, Φ-*array*, etc.) that can emulate powerful data structures, like suffix trees in a myriad of combinatorial string problems. These data structures play an important role in the efficient operation and low complexity of *Dfinder*; hence, in this chapter, we focus on their construction and their use in accelerating decision and enumeration queries.

The *suffix tree* (*ST*) [28] is a powerful data structure that has been used in many different areas such as in data compression [81], pattern matching [98], string processing [2], computational biology [85], etc. It is a compacted *trie* [10], refined to a minimum state finite automaton, that stores the suffixes of a sequence, so that all its possible subsequences are represented by a path originating from the root of the structure. Encoding all suffixes of a sequence in linear space allows the efficient retrieval of a large amount of information, such as the detection of patterns. Such pattern matching applications consist of detecting all occurrences of a subsequence in a sequence. However, *STs* suffer from two major factors that can seriously affect their performance: (i) although they have asymptotically linear space complexity, their memory consumption is quite large, and (ii) they usually suffer from poor locality in memory, achieving a mediocre caching utilisation. These drawbacks are easily observable in large applications that need to conduct a lot of queries using the *ST*.

The *SAs* were introduced by Manber and Myers [59] as a space-efficient alternative to *STs*, which also improves the execution time significantly. Additionally, it has been shown [1] that every algorithm that uses a *ST* can be replaced by an equivalent *SA*-based one, with the same time complexity.

3.1 Notation and Preliminaries

Before diving into the *enhanced SAs* and their construction, we are presenting some basic definitions and fundamentals of string processing and suffixes, that will ease the discussion.

Definition 3.1.1. An *alphabet* Σ is a finite ordered set of symbols, which has a fixed size $\sigma = |\Sigma|$, and the included symbols are comparable to each other.

Definition 3.1.2. A *sequence* T of length n is considered valid if it is a finite sequence of symbols over a given alphabet Σ ($T \in \Sigma^*$), with $T = t_0 t_1 ... t_{n-1}$. The empty sequence has length 0 and is denoted by ε.

Definition 3.1.3. Let T be a sequence of length n and i an index over it ($0 \leq i < n$). The notation t_i denotes the ith symbol of the sequence [1] . Moreover, given two indexes i and j, with $i \leq j$, we define the subsequence $T_{i...j}$ of T, that contains all symbols of T that are between the ith and the jth ones, including these two. However, if $i > j$, then $T_{i...j} = \varepsilon$, as such a subsequence cannot be defined.

Definition 3.1.4. A sequence of length n has n distinct suffixes and each such i^{th} suffix is denoted by $S_i = T_{i...n-1}$ and is identified by its starting position in the sequence (simply the index i).

For instance, in the context of a byte-level *differencing algorithm*, such as *Dfinder*, the alphabet Σ contains all possible values a single byte can get ($|\Sigma| = 2^8$), and the general comparison operation can be applied on them. Besides, each binary file can be interpreted as a sequence over Σ, and its size in bytes will be the size n of that sequence. Moreover, each such sequence T of length n, has n suffixes and the ith suffix is basically a subsequence of T, containing the last $n - i$ bytes ($S_i = t_i t_{i+1} ... t_{n-1}$, with $i \in [0, n)$) (see Figure 3.1).

Definition 3.1.5. The lexicographic comparison of two sequences U and V with size l_U and l_V, respectively, can be performed as follows:

- $U \underset{LEX}{=} V$ if $l_U = l_V$ AND $U_i = V_i \ \forall i \in [0, l_U)$
- $U \underset{LEX}{<} V$ if $U_i = V_i \ \forall i \in [0, min(l_U, l_V))$ AND $l_U < l_V$
  OR if $\exists \ j \in [0, min(l_U, l_V)) : U_j < V_j$ AND $U_i = V_i \ \forall \ i \in [0, j)$

One can easily observe that two suffixes of the same sequence can never be equal, as they inherently have unequal lengths. Moreover, notice that by combining the above operators, we can define more complex ones, such as $\underset{LEX}{\leq}$, $\underset{LEX}{\geq}$, etc.

Definition 3.1.6. The *longest common prefix* (*lcp*) of any two sequences U and V, can be computed as $lcp(U, V) = max\{\lambda \geq 0 : U_{0...\lambda-1} \underset{LEX}{=} V_{0...\lambda-1}\}$.

[1] Throughout this work, we consider that the indexing of all arrays starting at 0. Hence the 0th symbol corresponds to the first symbol of a sequence.

The last definition also applies at suffixes, taking into account their starting positions in the sequence. Namely, the *lcp* of two suffixes S_i and S_j of a sequence T is denoted by $lcp(i,j) = max\{\lambda \geq 0 : T_{i \dots i+\lambda-1} \underset{LEX}{=} T_{j \dots j+\lambda-1}\}$.

S_0	banana
S_1	anana
S_2	nana
S_3	ana
S_4	na
S_5	a

Table 3.1: The suffixes of sequence *"banana"*

3.2 The Suffix Array

The *SA* of a sequence T is an array that stores a permutation of the starting positions of all its suffixes, in lexicographically ascending order, so that $S_{(SA_i)} \underset{LEX}{\leq} S_{(SA_{i+1})}$, $\forall i \in [0, n-1)$; basically, SA_i contains the starting position of the ith smallest suffix of T. This way, when we refer to the order of a suffix, we refer to the entry (index) of SA, where this suffix has been mapped to.

Besides, as suffixes are mapped in *SA* in lexicographical order, they form groups (intervals) that share common prefixes. For example, suffixes that have the same first symbol will be mapped to consecutive entries of the *SA*, and the same holds for the suffixes that share the same first two, or three symbols, etc. As shown in Table 3.2, suffixes S_5, S_3 and S_1 form one such group of suffixes, sharing the common prefix *"a"*. On the other hand, S_3 and S_1 compose another group, sharing the prefix *"ana"*. Besides, we can conclude that two suffixes that share the *lcp* will be mapped to adjacent entries in the *SA*. This way, in order to find the *lcp* that a suffix may have with any other suffix, we can simply visit the two adjacent suffixes of it in the *SA*, and bilaterally compute the *lcp* of the former suffix with these two, keeping the maximum value.

Additionally, we define the inverse array of the *SA*, called *ISA* ($ISA = SA^{-1}$) that contains the rank of each suffix of T. Respectively, $\forall i \in [0, n)$, ISA_i contains the rank of suffix S_i, or equivalently, the number of lexicographically smaller suffixes. It must be noted that the rank is also the index of *SA* where the suffix has been mapped to. Using both *SA* and *ISA*, makes feasible the efficient detection of suffixes that share the *lcp*, by simply visiting adjacent entries in the *SA*. Namely, the rank of suffix S_i can be obtained as $i' = ISA_i$; thus, the suffixes that share the *lcp* with S_i can either be the one mapped in $SA_{i'-1}$ or $SA_{i'+1}$. Having precomputed only these two arrays, one has to perform an on-line comparison of the sequences to find which one shares the *lcp* with S_i.

Nevertheless, when many such queries are required during algorithm execution, *SA* and *ISA* are often used in conjunction with an auxiliary array, called *LCP array* (*LCPA*), which contains the length of the lcp between each pair of consecutive suffixes in the *SA*. More specifically, $LCPA_i$ contains the length of the lcp of the ith smallest suffix with the $(i-1)^{th}$ smallest one, $\forall i \in (0, n)$, while $LCPA_0 = 0$. Using the *LCPA* in conjunction with the *SA*, reduces the time complexity for deciding if a pattern P of length m exists in a sequence T of length n, from $O(mlogn)$ to $O(m + logn)$. In Table 3.2, we present the three aforementioned arrays, computed on the sequence *"banana"*. Having pre-computed the *LCPA*, one can calculate the *lcp* of any two suffixes S_i and S_j in linear time complexity, using Equation 3.1.

$$\begin{gathered} lcp(i,j) = min(LCPA_{floor+1}, \cdots, LCPA_{ceil}), \\ \text{with } floor = min(ISA_i, ISA_j) \text{ and } ceil = max(ISA_i, ISA_j) \end{gathered} \tag{3.1}$$

Index	SA	ISA	LCPA
0	5 ($S_5 : a$)	3	0
1	3 ($S_3 : ana$)	2	1
2	1 ($S_1 : anana$)	5	3
3	0 ($S_0 : banana$)	1	0
4	4 ($S_4 : na$)	4	0
5	2 ($S_2 : nana$)	0	2

Table 3.2: The SA, ISA and LCPA computed on sequence *"banana"*

3.3 SA and LCPA Construction

The efficient construction of *SAs* is a task that is accomplished through the application of a *suffix array construction algorithm* (*SACA*) [76] on the input sequence. If the sequence size is small, the *SA* can be computed naively, using a sorting algorithm, such as *bucket sort*, or *IntroSort*, to name a few. The latter sorting algorithm is a hybrid sorting algorithm that uses three sorting algorithms to achieve better execution time, *Quicksort*, *Heaport* and *Insertion Sort*. However, when the input sequence is larger, more advanced *SACAs* are needed, in order to achieve better execution time and keep the memory footprint low.

The first such algorithm is presented in the original paper of *SAs* [59] and is based on an iterative technique, called *prefix doubling*, having $O(nlogn)$ time complexity. The algorithm needs $logn$ iterations and operates as follows. Initially, all suffixes are put into groups (called buckets), according to their first symbol, using a sorting algorithm, such as bucket sort or radix sort, composing a preliminary *SA*. Inductively, during each iteration the buckets are further partitioned, sorting the respective suffixes according to twice the number of symbols that were scanned

during the previous iteration. This operation can be performed efficiently, utilising the *SAs* that were computed in the previous iterations. Furthermore, as soon as the algorithm finishes all suffixes are ordered, with each one of them having a distinct rank. Although this algorithm had super-linear (higher than linear) time complexity, at the time it was a space-saving alternative to the linear-time *ST* construction algorithms, which required a lot of space for the storage of the structure.

Later on, more works were presented [41, 31, 46, 44], proposing *SACAs* that achieved linear time complexity using different approaches, including *DC*, the ancestor of *DC3* [42]. However, all of them followed a recursive strategy for the construction of *SA*. Although the theoretical challenge was solved, it turned out that better tuned super-linear algorithms could still outperform the linear ones in practice. The reason for it is the large constant factors hidden in the asymptotic analysis of the linear algorithms, as well as their larger space requirements compared to the simpler super-linear counterparts. However, this changed in 2009 when the *SA-IS* algorithm [70] was introduced; an algorithm that is linear in theory and also very fast in practice. Later on, Yuta Mori, the creator of *DivSufSort* [2] further engineered *SA-IS*, improving its performance significantly. These two algorithms (*SA-IS* and *Divsufsort*) are currently considered the fastest suffix sorters. Since then, the competition in the field of suffix sorting has become very challenging. Some experimental *SACAs* have also been proposed that introduce new algorithmic ideas, such as *Radix SA* [77]. Finally, Bucket Pointer Refinement (BPR) [80] is another *SACA* that has been benchmarked multiple times in previous studies, showing that it performs well on real-world data and is remarkably fast on highly repetitive inputs.

As mentioned above, *SAs* are usually used in conjunction with *LCPAs*, allowing efficient queries to be conducted on the input sequence. The construction of the *LCPA* is usually possible during the computation of the *SA*, and many algorithms have been adapted accordingly; however, this is not always an easy task. Hence, apart from suffix sorting algorithms with integrated *LCPA* construction capabilities, have also emerged several "standalone" *LCPA* construction algorithms that require the input sequence and the pre-computed *SA* to construct the *LCPA*. In [59], the authors present an *LCPA* construction algorithm that has $O(nlogn)$ time and and $O(n)$ space complexity. Later on, in 2001, Kasai et al. presented *KLAAP* [43] (see Algorithm 1), the first linear algorithm that given an *SA* and the input sequence, can compute the *LCPA*. This algorithm is optimal in theory and also very fast in practice, while it required minimal space, for storing the *ISA*. Besides, Manzini et al. [62] have proposed a modified version of *KLAAP* that is more space-conscious, but in the experimental evaluation was shown that this version is up to two times slower than the original one. In 2009, Kärkkäinen, Manzini and Puglisi presented a refinement of KLAAP, called Φ, that achieved better execution time. Besides, instead of building the actual LCPA, the algorithm builds

[2] www.github.com/y-256/libdivsufsort

a permutated LCPA, in which the values appear in sequence order, instead of lexicographical order.

Algorithm 1 Kasai's LCPA construction algorithm (KLAAP)

```
    Input: SA, ISA, T
    Output: LCPA
 1: k ← 0
 2: i ← 0
 3: while i < n do
 4:     if ISA_i = 0 then
 5:         LCPA_(ISA_i) ← 0
 6:     else
 7:         j ← SA_(ISA_i − 1)
 8:         while T_(i+k) = T_(j+k) do
 9:             k ← k + 1
10:         end while
11:         LCPA_(ISA_i) ← k
12:         k ← max(k − 1, 0)
13:     end if
14: end while
```

Nevertheless, advances in suffix sorting algorithms soon emerged in a situation where the suffix sorting was faster than the construction of the *LCPA*. This situation is paradoxical as suffix sorting is considered to be a more computationally intensive operation than the computation of the *LCPA*. Hence since then, the two major suffix sorters have been augmented with the ability to also induce the *LCPA*, while computing *SA* [26].

During *Dfinder* development, we benchmarked some well-known *SACAs*, to select the one with the better performance, regarding the execution time and memory consumption, and integrate it in our algorithm. In order to conduct these experiments, we used *Saca-bench* framework [7, 6], which provides a plethora of *SACAs* implemented in C/C++ and an API for including additional *SACAs* in the tests. Based on our evaluation results, *DivSufSort* [60] was among the ones with the smallest execution time, while it also had a remarkably small memory footprint. These results keep in track with the ones presented in [77] and other benchmarks [3]. The results of our experiments about the execution time and memory consumption of the benchmarked *SACAs* are shown in Figure 3.1 and Figure 3.2. In order to produce the results we created a large number of files with varying sizes, and applied each algorithm, tracking the corresponding performance. Additionally, in order to get more accurate results, we repeated each experiment 100 times. Finally, besides *SACAs* we also tested a naive computation of the *SA* which is based on the *sort* function found in the standard library of C++ (std::sort)[4], but has quite large execution time.

[3] `www.github.com/y-256/libdivsufsort/blob/wiki/SACA_Benchmarks.md`
[4] std::sort is based on Introsort

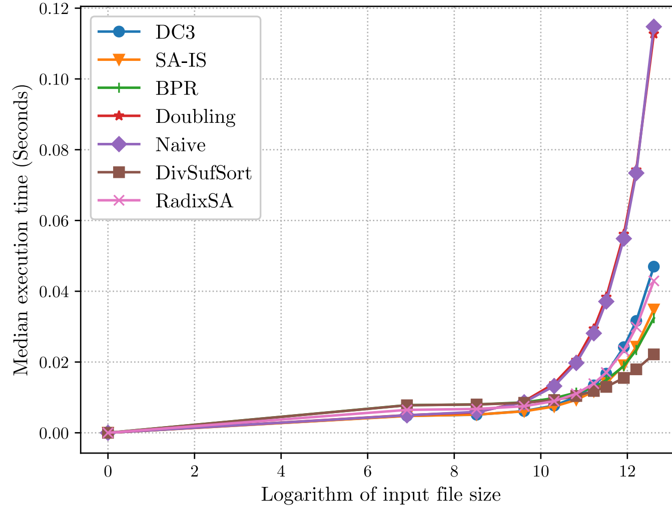


Figure 3.1: Median execution time of SACAs versus input file size[5]

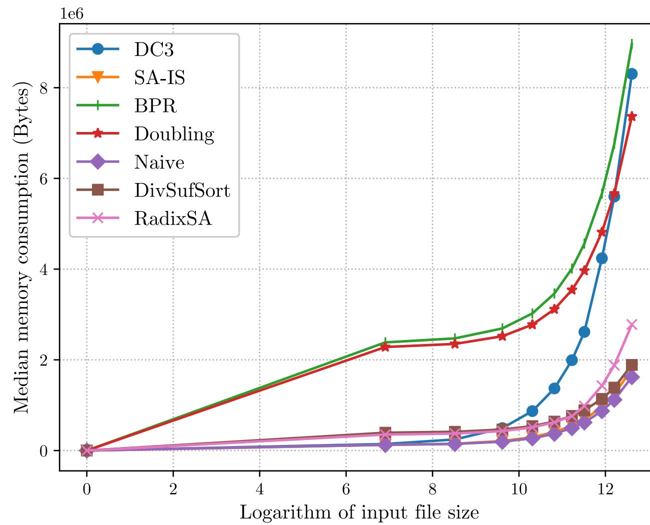


Figure 3.2: Median memory consumption of SACAs versus input file size

Finally, we decided to integrate *DivSufSort* (release v2.0.2-1) in *Dfinder*, due to its remarkable performance, as well as the vast attention it has received from the

[5]The system used to conduct the experiments had an Intel® Xeon® CPU E5-2630 v2 clocked at 2.60GHz, with 4 GiBs of RAM and Linux kernel 5.4.0-70-generic.

research community, resulting in its thorough validation, porting to various programming languages, and implementation of parallel *DivSufSort* versions. These features render *DivSufSort* an ideal option for *Dfinder*, as seamless porting to other platforms is feasible, a trait that can further improve the performance and scalability of our algorithm in the future. Besides, we chose to integrate a modified version of *DivSufSort* (presented in [26]), that also induces the *LCPA*, as based on the benchmarks presented in [26] it is the optimal method for computing both the *SA* and *LCPA*, with minimal memory overhead.

3.4 DivSufSort

DivSufSort is currently considered a state-of-the-art suffix sorter and has been integrated into many works, such as bioinformatics libraries [6]. Although it is not a linear (in time) algorithm (it has $O(nlogn)$ time and $O(n)$ space complexity), it is very fast in practice, as it avoids the overhead of recursions and fully utilises *induced sorting*; a sorting principle that induces the order of suffixes using a subset of suffixes that have already been sorted, by prepending symbols. Besides, the algorithm is very space-conscious, as it needs very little space mainly for storing bucket pointers. Although *DivSufSort* is open-source, due to its complication, the first documentation of it was introduced in 2017 [26]. In the same work, the authors augmented the algorithm, allowing it to also induce the *LCPA* in conjunction with the *SA*; this *DivSufSort* version we further augmented and integrated into *Dfinder*. In this section, we main components and operation of the *DivSufSort* algorithm, based on the documentation provided in [26].

3.4.1 Additional Notations

In addition to the definitions provided in Section 3.1, we introduce some suffix classifications, needed for the comprehension of *DivSufSort*. A suffix S_i is an *A-suffix* if $t_i > t_{i+1}$ or if $i = n - 1$. On the other hand, if $t_i < t_{i+1}$ the suffix is considered to be a *B-suffix*. Last, if $t_i = t_{i+1}$, then the suffix S_i is assigned to have the same type as the its next one (S_{i+1}). Finally, if a suffix S_i is a *B-suffix* and suffix S_{i+1} is an *A-suffix*, then S_i is also a *B*-suffix*. This classification results in a partial order among the suffixes. For instance, an *A-suffix* will be lexicographically smaller than a *B-suffix*, if they share the same first symbol. Finally, given two consecutive *B*-suffixes* S_i and S_j, the string $T_{i..j+1}$ is called *B*-substring*.

3.4.2 Basic Operation

DivSufSort consists of three stages. In the first stage, the algorithm determines the type of all suffixes (if they are A-, B-, or/and B*-suffixes) in a single scan of the sequence from right to left. Moreover, during this stage, the algorithm composes

[6] `www.github.com/NVlabs/nvbio`

the c_0- and (c_0, c_1)-buckets, assigning suffixes into buckets according to their first, or first two symbols, respectively, and the borders of the computed buckets are recorded.

During the second stage, all *B*-suffixes* are sorted and put in the SA; this is a three-step process. First, all *B*-substrings* are sorted in-place and independently for each (C_0, C_1) bucket. The sorting is performed using an implementation of *IntroSort* (ISS) [67], that uses the *Multikey Quicksort* (*MKQS*) [9] and *Heapsort* (*HS*). After all *B*-substrings* have been ordered, the *ISA* is computed on them, which will contain the rank of each *B*-suffix*. Then, using *ISA*, the ranks of all *B*-suffixes* are computed, following an approach similar to prefix doubling. However, instead of doubling the length of the suffix prefix that is examined in each iteration, the algorithm doubles the number of the considered *B*-substrings*, which can have an arbitrary length. At this point, the algorithm has computed the ranks of all *B*-suffixes*. Their sorting is performed using Quicksort (QS), augmented by the *repetition detection* [61] technique. In this way, the final *ISA* of all *B*-suffixes* is computed and hence the *SA* can be calculated in linear time complexity.

During the last stage, the algorithm induces the *A-suffixes* and *B-suffixes* using the partial order that exists in any (C_0, C_1) bucket; *A-suffixes* are lexicographically smaller than *B-suffixes* and *B*-suffixes* are smaller than *B-suffixes*. The algorithm scans the computed *SA* twice. First, scanning from left to right the *A-suffixes* are induced, while the *B-suffixes* are induced scanning the opposite direction (from right to left).

3.4.3 Inducing the ISA and LCPA, in conjunction with the SA

The authors in [26], apart from the documentation of *DivSufSort*, also proposed a modification that allows it to induce the *LCPA*, in conjunction with the *SA*. As soon as the *B*-suffixes* are lexicographically ordered, *LCPA* is computed on them using a sparse version of the Φ-algorithm [52]. On the other hand, *A-suffixes* and *B-suffixes* are induced using a technique that allows answering *range minimum queries* (RMQs) using a min-stack data structure. Finally, the algorithm computes the *LCP* values of the suffixes that are placed at bucket borders, as they cannot be automatically induced. This approach was shown [26] to be faster than all the previous *LCP* inducing *SACA*s based on *SA-IS* and competitive with the Φ-algorithm, the fastest known "standalone" *LCP* construction algorithm. Finally, we modified the algorithm, so that it also induces the *ISA*, as it is computed during the final iteration. However, we did not measure any significant improvement on the performance, as the *ISA* construction is a computationally lightweight operation, given the precomputed *SA*.

Chapter 4

Dfinder

Delta encoding aims at producing a delta script, which expresses the differences between two file versions; in the context of this thesis, between two firmware versions. Interpreting the delta script, the receiver can reconstruct the updated firmware version locally, utilising (copying) segments of the current one. *Dfinder* is a byte-level differencing algorithm that follows a dynamic programming approach, which allows the efficient computation of small delta scripts. Its competitive advantage, over other such algorithms, is that using the *Block Moves Detector* (*BMD*) module (discussed in section 4.1), the algorithm can detect various types of matching segments, for each segment of the updated firmware (see Table 4.1). This trait allows it to increase the ratio of the bytes of the updated firmware, which can potentially be encoded using *COPY* instructions, effectively reducing the size of the final delta script. More precisely, for each segment of the updated firmware *BMD* can detect matching ones in the current firmware, which can be copied either in forward or reverse order to reconstruct the former. For instance, *BMD* considers both *"abcde"* and *"edcba"* (found in the current firmware) as matching ones for segment *"abcde"* of the updated image. Besides, *BMD* can utilise parts of the updated firmware that have already been reconstructed by the receiver to find matching segments for following segments (of the updated firmware). These segments found in the par-

Type 1	Matching segments found in the current firmware, which must be copied in forward order.
Type 2	Matching segments found in the current firmware, which must be copied in reverse order.
Type 3	Matching segments found in the partially reconstructed updated firmware, which must be copied in forward order.
Type 4	Matching segments found in the partially reconstructed updated firmware, which must be copied in reverse order.

Table 4.1: Types of matching segments *BMD* is able to detect for each segment of the updated firmware

tially reconstructed part of the updated firmware can be copied either in forward or reverse order to reconstruct respective following segment.

The encoding of delta scripts that *Dfinder* outputs, supports two prime instructions: *ADD* and *COPY*. Besides, *COPYs* are further subdivided to four sub-instructions ($Copy_typeX | X \in \{1, 2, 3, 4\}$), according to the type of the matching segment that *BMD* has detected for the encoded one, such as the ones presented in Table 4.1. Although the same format is used for each such *COPY* variant, a distinct opcode is required (1,2,3 and 4, respectively), so that the type can be recognized by the receiver and the reconstruction of the encoded segment can be performed properly. The format of these instructions is shown in Figure 4.2 and Figure 4.1, with overhead of $\alpha = 2$ bytes for *ADDs* and $\beta = 5$ bytes for *COPYs*, needed for containing the opcode, the length of the encoded segment, etc. In the final delta script, consecutive bytes of the updated firmware form regions of bytes, which can be reconstructed copying their matching segments, and are encoded using *COPY* instructions, while the rest need to be explicitly transmitted and are encoded using *ADDs*. As each *ADD* instruction has an non-negligible overhead, consecutive *ADDs* are merged to form larger ones. Hence, although two *COPYs* may be placed one after the other in the final delta script, the same does not hold for two (or more) *ADDs*.

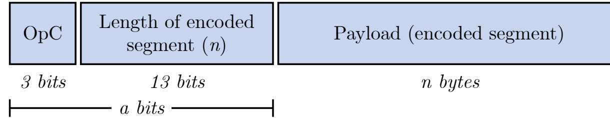


Figure 4.1: *ADD* instructions format

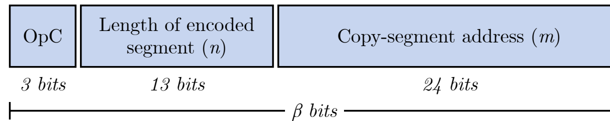


Figure 4.2: *COPY* instructions format

Each byte of the updated firmware can be encoded using an *ADD* instruction, while some may also be able to be encoded using one or more possible *COPYs*, if they belong to segments of bytes for which *BMD* was able to detect matching ones. The goal of the algorithm is to select the optimal instruction, which can encode each such byte of the updated firmware, resulting in the minimisation of the final delta script size. The worst-case scenario which results in the largest possible delta script, is when all bytes of the updated firmware (of size n) are encoded using a single *ADD* instruction, causing a delta script of size $\alpha + n$ bytes. On the other hand, in the case of two identical firmware images, all bytes will form a segment encoded in a single *COPY*, resulting in a delta script of β bytes. Actually, the last two examples are not entirely accurate regarding the delta script

size, as in our implementation each encoded segment is limited to $2^{13} - 1$ bytes, as the field that contains the length of it, is 13 bits long. Based on our experiments, this size is sufficient, as we did not find matching segments with length larger than that threshold for actual firmware images. Besides, apart from the length of the encoded segment, a *COPY* instruction also contains the initial address of the *Copy-Segment*, which is the segment that needs to be copied to reconstruct the encoded one. As mentioned earlier, the former segment can either be found in the current firmware image, or the partially reconstructed part of the updated image. Hence, the address of this segment (m) is represented as the distance of it from the beginning of the respective image. As the receiving node is aware of the initial address of these images, the reconstruction of the updated firmware is seamless and does not require additional information (e.g. absolute flash addresses).

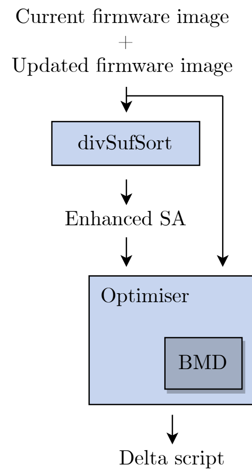


Figure 4.3: The main components of *Dfinder*

The main structure of *Dfinder* is shown in Figure 4.3. The two firmware images are concatenated and the resulted byte-array is given as input to *divSufSort*, so that it can compute the required *enhanced SA*. Then, *Optimiser* (see Section 4.2) uses the resulted data structures, as well as the firmware images, to compute the delta script. To this end, *Optimiser* uses *BMD*, in order to detect matching segments and select the optimal encoding-instruction for each byte of the updated firmware, composing the final delta script as a combination of consecutive *ADD* and *COPY* instructions.

4.1 Block Moves Detector (BMD)

The *string to string correction problem* was defined in [95], as the problem of finding the optimal combination of operations (insert, delete and update) that transform a sequence into another. The first approaches to solve this problem were based on finding the longest common subsequences between two sequences, using

dynamic programming. However, these solutions fail to capture the generality of delta compression. For instance, they assume that the common subsequences are found in the same order in the two sequences, and are not able to capture repetitive subsequences.

In order to resolve these limitations, Tichy defined the same problem using *block moves* [91]. A block move is a triple (m, k, λ) which describes the bounds of two subsequences of length λ, originating at the mth and k^{th} symbol (e.g. byte) of the respective file version, so that $OLD_{m...(m+\lambda-1)} \underset{LEX}{=} NEW_{k...(k+\lambda-1)}$. Hence, a delta script can be constructed as a minimal covering set of such block moves that describe overlapping matching segments between two file versions; *BMD* is used for finding such block moves between two firmware versions (*Current* and *Update*), as well as between the partially reconstructed updated firmware and the rest of it. For this purpose, *Dfinder* constructs the byte-array T (see Figure 4.4), as a composition of the two firmware versions in both forward and reverse order, and the *enhanced SA* (*SA*, *ISA* and *LCPA*) is calculated based on it, using *divSufSort*.

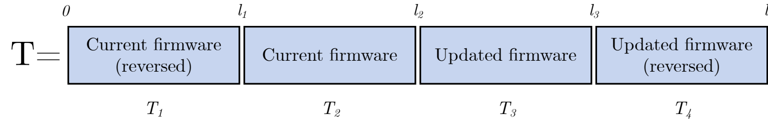


Figure 4.4: The construction of byte-array T as a composition of the two firmware versions, the current and the updated one

Given an index i, that resembles the ith byte of the updated firmware, *BMD* initially computes its' mapping index i' in T, as $i' = l_4 - i - 1$ and obtains the suffix $S_{i'}$, which originates in T_4. The main goal is to find the suffix $S_{j'}$, which shares the *lcp* with $S_{i'}$, and also fulfills the requirements for at least one of the possible matching segments types that are discussed below. This way, the common prefix of these two suffixes, when mapped to the actual firmware images, resembles the block move (m, k, λ), which allows the reconstruction of $Update_{k...i}$, with $i = k + \lambda - 1$. Besides, *BMD* tries to find the smallest possible k, so that it results in the largest possible encoded segment. This block move can also be interpreted as a pair of matching segments either (i) between the two firmware versions, or (ii) between the partially reconstructed updated firmware and the remaining part of the updated firmware. The first segment of this pair is a segment of the updated firmware that could potentially be reconstructed, copying the second one, either in forward or reverse order. We refer to the former segment as *Reference-Segment* (*RS*) and the latter as *Copy-Segment* (*CS*).

Using the pre-computed *enhanced SA*, *BMD* can find the suffixes that share the *lcp* with $S_{i'}$ very efficiently, as they are placed in consecutive entries of *SA*. In such a way, the algorithm iteratively visits the suffixes that have been placed in increasingly further entries of *SA*, from the one where $S_{i'}$ has been mapped to, and it finally stops when it finds a suffix that fulfills the requirements of at least

one of the matching segments types. Namely, if $S_{i'}$ is the cth smallest suffix of T, it will be mapped to the cth entry of *SA*. *BMD* will iteratively visit $SA_{c\pm1}$, $SA_{c\pm2}$, $SA_{c\pm3}$, ..., until it finds the suffix $S_{j'}$, which satisfies the requirements of at least one matching segments type (see Algorithm 2). The common prefix of these two suffixes resembles a pair of matching segments -or a block more-, when mapped back to the domain of the actual firmware images. Finally, if *BMD* was not able to find a suitable block move for given i, it denotes this inability returning $k = \infty$.

In order to ease the discussion about *BMD* we provide some further clarifications and definitions. First, we define the function *map*, which allows the transition from the domain of T to the one of the respective firmware image. This function gets an index i' and returns the index i, so that $T_{i'} = Firmware_i$, where *Firmware* is the respective firmware image which is accommodated in the region of T (T_1, T_2, T_3, or T_4), where i' is located. Namely, if $l_2 \leq i' < l_3$, i will be the mapping index in the domain of the updated firmware, so that $T_{i'} = Update_i$. Moreover, if the firmware image in that region is in its original order (not reversed), i can simply be computed by the order of i' in that specific region. However, if the image is reversed (e.g. T_4), i is calculated as the mirroring index of i' in that region. For instance, if i' is the second in order index in T_4, i is the second to last index of the updated firmware. This also implies that if two indices i' and j' are located within a region where the corresponding firmware is reversed and $i' < j'$, the relationship of their mapping indices i and j in the respective firmware image, will be reversed ($i > j$) and vice versa. The function *map* also enables the transition of a subsequence of T to the domain of the respective firmware image, simply using the function on the bounds of that sequence. For instance, if two indices i' and j' are within $[l_1, l_2)$ with $i' < j'$, we can conclude that $T_{i' \ldots j'} \underset{LEX}{=} Current_{i \ldots j}$ ($i < j$). However, if i' and j' were within $[0, l_1)$, the resulted subsequence $Current_{i \ldots j}$ would correspond to ε, as the firmware that is accommodated in that region is reversed and hence $i' < j' \iff i > j$. For this reason we introduce the function *reverse*, which reverses the order of symbols found in a sequence. This way, we can proceed with the previous example, having that $T_{i' \ldots j'} \underset{LEX}{=} reverse(Current_{j \ldots i})$ ($j < i$).

4.1.1 Type 1: Matching segments found in the current firmware (copied in forward order)

The first region of T (T_1) is used for finding *CSs* in the current firmware version, which can be copied in forward order to reconstruct the respective *RS*. In order such a *CS* to be valid, the *lcp* of $S_{j'}$ and $S_{i'}$, that originates at j', has to be entirely within T_1. As shown in Figure 4.5, the subsequences $T_{j' \ldots m'}$ and $T_{i' \ldots k'}$ are equal, as they correspond to the *lcp* of $S_{j'}$ and $S_{i'}$. In order the subsequence $T_{j' \ldots m'}$ to be within T_1, m' has to be less than l_1, with $m' = j' + \lambda - 1$. Moreover, as the pair of indices (j', m') is within T_1 and (i', k') within T_4, which both regions accommodate reversed images, we can imply that their mapping indices to the actual firmware images (current and update) will have reversed relationship: $j > m$ and $i > k$.

Hence, having that $T_{j'\dots m'} \underset{LEX}{=} T_{i'\dots k'}$ results in $Current_{j\dots m} \underset{LEX}{=} Update_{i\dots k}$ ($i > k$, $j > m$), and applying the *reverse* function to both sides of the equation, results in $Current_{m\dots j} \underset{LEX}{=} Update_{k\dots i}$, with $m < j$ and $k < i$. Hence, the segment of the current firmware shown at the left side of the equation can be copied as is to reconstruct the corresponding segment $Update_{k\dots i}$ of the updated firmware.

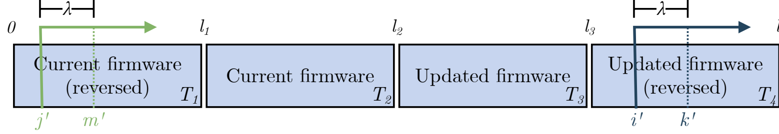


Figure 4.5: Finding *Type 1* matching segments

4.1.2 Type 2: Matching segments found in the current firmware (copied in reverse order)

Similar to the previous matching segments type, T_2 is used for finding *CSs* in the current firmware, which need to be copied in reverse order to reconstructs the corresponding *RS*. For this matching segments type to hold, the *lcp* of $S_{j'}$, and $S_{i'}$, originating at j', has to be within the bounds of T_2, as shown in Figure 4.6; in other words, $j' > l_1$ and $m' < l_2$, with $m' = j' + \lambda - 1$. This way, the segment $T_{j'\dots m'}$, when mapped to the current firmware, fits entirely in the image.

Having that $T_{j'\dots m'} \underset{LEX}{=} T_{i'\dots k'}$, with $j' < m'$ and $i' < k'$, and given that only the pair of indices (i', k') is in region that accommodates a reversed image, we can conclude that for the mapping indices: $j < m$ and $i > k$. Additionally, using the *map* function on the bounds of the two segments, results in $Current_{j\dots m} \underset{LEX}{=} Update_{i\dots k}$. Furthermore, applying the *reverse* function on both sides of the last equation, we get that $reverse(Current_{j\dots m}) \underset{LEX}{=} Update_{k\dots i}$, with $j < m$ and $k < i$. Thus, in order to reconstruct the segment $Update_{k\dots i}$, the segment $Current_{j\dots m}$ has to be copied in reverse order (originating from the mth byte of the current firmware and finalizing at the jth one, with $j < m$).

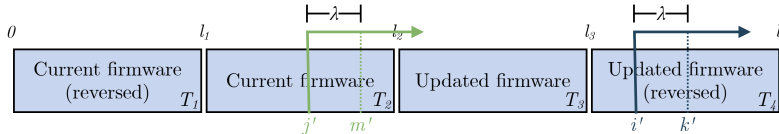


Figure 4.6: Finding *Type 2* matching segments

4.1.3 Type 3: Matching segments found in the partially reconstructed updated firmware (copied in forward order)

Apart from containing the reference suffix $S_{i'}$, T_4 is also used for finding *CSs* in the partially reconstructed part of the updated firmware, which need to be copied in forward order to reconstruct the respective *RSs*. As the node interprets the delta script, it will progressively reconstruct the updated firmware. Hence, when a segment of the updated firmware is about to be reconstructed, the updated firmware up to this point will have already been reconstructed and be available for finding *CSs* at, as they precede the respective *RSs* in order.

The main requirement for this matching segments type is the chosen *CS* to always precede the respective *RS* in order. As the *CS* ($Update_{m...j}$) has to precede the *RS* ($Update_{k...i}$) in the domain of the updated firmware, the prime requirement of this matching segments type is $k > j$, so that the former segment entirely precedes the latter in order. Given that all indices (m', j', k', i') are within T_4, which contains a reversed firmware image, we can also conclude that $k > j \iff k' < j'$. Thus, the suffix $S_{j'}$ can be used for finding *Type 3* matching segments, only if j' is greater than the respective k'.

Having that $T_{j'...m'} \underset{LEX}{=} T_{i'...k'}$, being the *lcp* of $S_{j'}$ and $S_{i'}$, and that all pairs of indices are within T_4 (see Figure 4.7), we can conclude that $Update_{j...m} \underset{LEX}{=} Update_{i...k}$ with $j > m$ and $i > k$. Thus, applying the *reverse* function on both sides, the above equation results in $Update_{m...j} \underset{LEX}{=} Update_{k...i}$. Finally, the segment of the updated firmware at the left side of the equation will always precede the one at the right side, as $k > j$, and hence could be copied to reconstruct it.

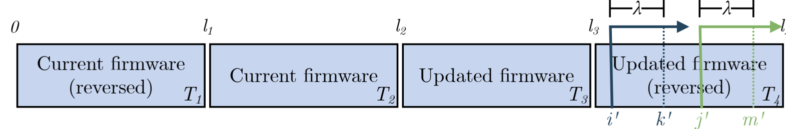


Figure 4.7: Finding *Type 3* matching segments

4.1.4 Type 4: Matching segments found in the partially reconstructed updated firmware (copied in reverse order)

In contrast to the previous matching segments type, the segments of the partially reconstructed update firmware that have to be copied in reverse order to reconstruct the respective *RSs* can be found in T_3. In order to ensure that the *CS* precedes the *RS* in the domain of the updated firmware, the value z' is computed as the mirroring index of k' in T_3 ($z' = 2 * l_3 - k'$). Moreover, as the indexes m' and z' are in a region with non-reversed firmware image, we conclude that their mapping indices to the updated firmware image keep their relationship: $m' < z' \iff m < z$. Furthermore, the mapping index of k' in the updated firmware, k equals to z, because z' was computed as the mirroring index of k' in T_3. Hence, as the *CS*

($Update_{m \dots j}$) has to precede the *RS* ($Update_{k \dots i}$) in the domain of the updated firmware, $m < k \iff m' < z'$. The previous concludes that as $T_{j' \dots m'} \underset{LEX}{=} T_{i' \dots k'}$, the former subsequence has to be within $[l_2, z')$, as shown in Figure 4.8.

After applying the *map* function on both sides of equation $T_{j' \dots m'} \underset{LEX}{=} T_{i' \dots k'}$, results in $Update_{j \dots m} \underset{LEX}{=} Update_{i \dots k}$, with $k < i$ and $j < m < z$ (thus $j < m < k < i$). Using the *reverse* function on both sides of the equation, we conclude that $reverse(Update_{j \dots m}) \underset{LEX}{=} Update_{k \dots i}$. Hence, in order to reconstruct the segment $Update_{k \dots i}$, we can copy $Update_{j \dots m}$ in reverse order. Besides, it was shown above that $m < k$, ensuring that the latter segment entirely precedes the former.

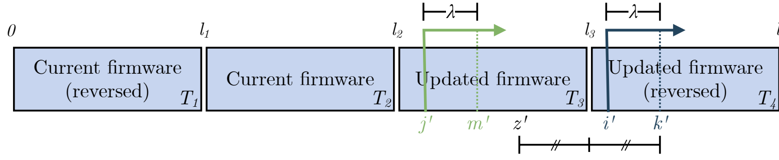


Figure 4.8: Finding *Type 4* matching segments

Algorithm 2 Block Moves Detector

Input: SA, ISA, $LCPA$, i
Output: m, k, λ

1: $i' \leftarrow l_4 - i - 1$
2: $offset \leftarrow 1$
3: $j' \leftarrow SA_{ISA_{i'}+offset}$
4: **while** No matching segment is found **do**
5: $temp_\lambda \leftarrow lcp(j', i', LCPA)$
6: **if** $temp_\lambda = 0$ **then**
7: *break*
8: **end if**
9: $m' \leftarrow j' + temp_\lambda - 1$
10: $k' \leftarrow i' + temp_\lambda - 1$
11: $z' \leftarrow 2 * l_3 - k' + 1$
12: **if** $0 < j' < l_1$ & $j' < m' < l_1$ **then** ▷ Check requirements for *Type 1* matching segments
13: $(R_j', R_m', R_k', R_\lambda, R_type) \leftarrow (j', m', k', temp_\lambda, 1)$
14: *break*
15: **else if** $l_1 < j' < l_2$ & $j' < m' < l_2$ **then** ▷ Check requirements for *Type 2* matching segments
16: $(R_j', R_m', R_k', R_\lambda, R_type) \leftarrow (j', m', k', temp_\lambda, 2)$
17: *break*
18: **else if** $j' > k'$ **then** ▷ Check requirements for *Type 3* matching segments
19: $(R_j', R_m', R_k', R_\lambda, R_type) \leftarrow (j', m', k', temp_\lambda, 3)$
20: *break*
21: **else if** $l_2 < j' < l_3$ & $j' < m' < z'$ **then** ▷ Check requirements for *Type 4* matching segments
22: $(R_j', R_m', R_k', R_\lambda, R_type) \leftarrow (j', m', k', temp_\lambda, 4)$
23: *break*
24: **end if**
25: $offset \leftarrow offset + 1$
26: $j' \leftarrow SA_{ISA_{i'}+offset}$
27: **end while**

28: ▷ Repeat lines 4-27 with $j' \leftarrow SA_{ISA_{i'}-offset}$ and decrementing offset in each iteration, calculating $(L_j', L_m', L_k', L_\lambda, L_type)$
29: **if** $R_\lambda > L_\lambda$ **then**
30: $m \leftarrow map(R_m')$
31: $k \leftarrow map(R_k')$
32: $\lambda \leftarrow R_\lambda$
33: **else**
34: $m \leftarrow map(L_m')$
35: $k \leftarrow map(L_k')$
36: $\lambda \leftarrow L_\lambda$
37: **end if**
38: **return** (m, k, λ)

4.2 Optimiser

Optimiser is the component of *Dfinder* which constructs delta scripts, aiming at minimizing their final size. To this end, *Optimiser* follows a dynamic programming approach, similar to the one presented in [19], which allows the selection of the optimal instruction (*ADD* or *COPY*), in order to encode each byte of the updated firmware.

At this point, we will introduce some notations which will allow further discussion about *Optimiser*. First, let Opt_i denote the size of the smallest partial delta script that needs to be transmitted so that the receiver can reconstruct the first $i + 1$ bytes of the updated firmware (of n bytes). We can conclude that (i) if $i > j$, then $Opt_i \geq Opt_j \; \forall i, j \in [0, n)$, and (ii) Opt_{n-1} represents the size of the final delta script, as it resembles the size of the smallest delta script, which allows the reconstruction of the entire updated firmware. Moreover, we introduce two additional notations: Opt_i^A and Opt_i^C. The former represents the size of the smallest partial delta script, which enables the reconstruction of the first $i+1$ bytes of the updated firmware, and the last instruction is an *ADD*. On the other hand, the latter notation represents the size of the smallest partial delta script, which enables the reconstruction of the first $i+1$ bytes of the updated firmware, and the last instruction is a *COPY*. Hence, Opt_i can be computed as the minimum value of these two notations ($Opt_i = min(Opt_i^A, Opt_i^C)$). Additionally, we introduce the notation BP_i (back-patch), which denotes if it is preferred the $(i-1)^{th}$ byte to be encoded using an *ADD* or a *COPY* instruction, because this information is available for a byte of the updated firmware only when the next (i^{th}) byte has been processed.

Optimiser consists of two primal stages. During the first stage, each byte of the updated firmware is processed individually, and the corresponding values are computed, progressively filling the Opt^A, Opt^C, Opt and BP arrays (see Algorithm 3). Starting from the 0th (first) byte of the updated image, these values are computed for each ith byte, until Opt_{n-1} is finally calculated. During this process, the value Opt_i^C is computed as $Opt_{k-1} + \beta$, if a respective block move was detected by *BMD* for the given i, or $Opt_i^C \leftarrow \infty$, otherwise. On the other hand, Opt_i^A is computed as $Opt_i^A \leftarrow min(Opt_{i-1}^A + 1, Opt_{i-1}^C + \alpha + 1)$.

Although during the $(i-1)^{th}$ iteration, *Optimiser* may choose to encode the $(i-1)^{th}$ byte using a *COPY* instruction, in the next (i^{th}) iteration it may infer that it is optimal to encode the last two bytes ($(i-1)^{th}$ and i^{th}) using an *ADD*. This transition of the encoding of the previous byte is flagged, using the *BP* bit-array. This array allows the efficient injection of *ADD* instruction in the delta script in the next stage, as segments which should be encoded using *ADDs* will form regions in *BP*, with value 1. It is computationally inexpensive to find such segments using *BP*, as the algorithm only needs to find the highest ordered byte j, for which: $Opt_j^A < Opt_j^C$. It can be proven that under the given encoding, this algorithm produces an accurate estimation of the size of the the optimal delta script [19].

Algorithm 3 Computation of the optimal arrays

Input: n
Output: Opt^A, Opt^C, Opt, BP

```
1: BP_0 ← 0
2: Opt_0^C ← β
3: Opt_0^A ← α + 1
4: Opt_0 ← MIN(Opt_0^A, Opt_0^C)
5: for i = 1 to n do
6:     k ← BMD(i).k
7:     if k = ∞ then
8:         Opt_i^C ← ∞
9:     else
10:        Opt_i^C ← Opt_{k-1} + β
11:    end if
12:    add_1 ← Opt_{i-1}^A + 1
13:    add_2 ← Opt_{i-1}^C + α + 1
14:    Opt_i^A ← MIN(add_1, add_2)
15:    BP_i ← ARGMIN(add_1, add_2)
16:    Opt_i ← MIN(Opt_i^A, Opt_i^C)
17: end for
18: return Opt^A, Opt^C, Opt, BP
```

Algorithm 4 Instructions creation helper functions

```
1: function CreateAddInstruction(length, segment)
2:     Instruction c
3:     c.opcode ← 0
4:     c.length ←length
5:     c.encodedSegment ← segment
6:     return c
7: end function

1: function CreateCopyInstruction(m, k, length, type)
2:     Instruction c
3:     c.opcode ← type
4:     c.m ← m
5:     c.k ← k
6:     c.length ← length
7:     return c
8: end function
```

During the next stage (see Algorithm 5), the bytes of the updated firmware are iterated over in reverse order, and the respective Opt_i^C and Opt_i^A are compared to infer if the optimal option is to inject an *ADD* or a *COPY* instruction in the

Algorithm 5 Computation of the delta script

Input: Opt^A, Opt^C, BP, $Update$
Output: Final delta script

```
 1: i ← n − 1
 2: pos ← 0
 3: while i ≥ 0 do
 4:     if Opt_i^A > Opt_i^C then
 5:         (m, k, λ, type) ← BMD(i)
 6:         Instructions_pos ← CreateCopyInstruction(m, k, λ, type)
 7:         i ← k − 1
 8:     else
 9:         l ← BP_i
10:         i_temp ← i − 1
11:         while l = 1 do
12:             l ← BP_i_temp
13:             i_temp ← i_temp − 1
14:         end while
15:         Instructions_pos ← CreateAddInstruction(i − i_temp, Update_i_temp...i)
16:         i ← i_temp
17:     end if
18:     pos ← pos + 1
19: end while
20: return Reverse(Instructions)
```

delta script. If the former is the case, the algorithm checks if there is a contiguous block of bytes in BP array with value 1, ending at the ith byte. This block, which is originated from a lower-ordered byte and finalizes at the currently iterated byte i will be encoded using a single ADD instruction and the next byte to check will be the one that precedes the one at the beginning of the block. On the other hand, if the algorithm detected that a $COPY$ command is preferred, a $COPY$ instruction is injected in the delta script, containing the information needed for the reconstruction of the encoded segment (starting from the kth byte and finalizing at the currently iterated byte). Again, the algorithm continues the reconstruction from the $(k-1)^{th}$ byte, which precedes the encoded segment. To construct the delta script, *Optimiser* uses the functions found in Algorithm 4, which allow the construction of ADD and $COPY$ instructions.

Proof that *Optimiser* computes the smallest delta script under the given encoding

First, $Opt_0^A = \alpha + 1$ and $Opt_0^C = \beta$ and represent the size of the delta script needed to reconstruct the first byte of the updated firmware, sending an ADD and a $COPY$ instruction, respectively. These values are optimal, as we know that in the first case the total size of the delta script that enables the reconstruction of the first byte is $\alpha + 1$ bytes and β bytes in the latter case. Additionally, if Opt_i^A and Opt_i^C are optimal for byte i, we can conclude that Opt_i is also optimal because it is calculated as the minimum value of them.

Using induction we want to prove that for every byte i, both opt_i^C and opt_i^A are optimal given that these values are optimal for all previous bytes (e.g. $i-1$).

Proving that opt_i^C is optimal, is quite simple. $Opt_i^C = opt_k + \beta$ and k is the smallest possible index, so that the segment $Update_{k...i}$ can be reconstructed by another segment. Besides, opt_k is optimal, because $k < i$. Hence, opt_i^C is optimal.

Regarding opt_i^A, it is calculated as $opt_i^A = min(opt_{i-1}^A + 1, opt_{i-1}^C + \alpha + 1)$. If the previous byte was added, then $opt_i^A = opt_{i-1}^A + 1$, as the last byte will be encoded in a larger *ADD* instruction. On the other hand, if the $(i-1)^{th}$ byte was copied ($opt_{i-1}^C \leq opt_{i-1}^A$), then $opt_i^A = \alpha + 1$, as a new *ADD* instruction must be created to encode the last byte. opt_i^A is calculated as the minimum cost with the last-encoded byte and hence it is optimal, given that both opt_{i-1}^A and opt_{i-1}^C are optimal.

4.3 In-place Reconstruction of the Updated Firmware

Traditional *incremental programming* approaches, based on differencing algorithms, require the receiver to have enough storage space, for accommodating both firmware versions, simultaneously. In this update fashion, the current firmware remains in flash, while the updated image is reconstructed at a different flash address. This approach allows the current firmware to stay intact so that segments of it can be copied for reconstructing the updated firmware.

Although this approach allows the algorithm to find the maximum number of such matching segments between the current and the updated firmware version, it may be infeasible as the receiver may not have enough storage, due to cost limitations. To this end, we propose an extension of *Dfinder*, which guarantees that the reconstruction can be performed *in-place*; starting at the same flash address, where the current version is located.

As the updated firmware is reconstructed at the same address where the current one is located, the current version will progressively be overwritten. As a result, *CS* which are located at this overwritten part may be altered, resulting in faulty reconstruction of the respective *RS*. Interestingly, this overwritten flash region will now accommodate the partially reconstructed part of the updated firmware, which is used for finding *type 3* and *type 4 CSs*. Hence, we have to guarantee that when the reconstruction is conducted in-place, the *type 1* and *type 2 CSs* will succeed the respective *RSs* in order, so that they are not yet overwritten.

For *type 1 CSs* this requirement is fulfilled, simply ensuring that $|l_1 - m'| \geq |l_4 - k'|$, so that for the mapping indexes in the two firmware versions: $m \geq k$. This way, when the segment $Update_{k...i}$ is about to be reconstructed, the respective *RS*, $Current_{m...j}$ will not be overwritten. Regarding *type 2 CSs*, we cannot use their initial address m for ensuring the aforementioned requirement, as these segments are copied in reverse order, and m is the highest offset of that segment ($reverse(Current_{j...m})$). Hence, purely based on m in this case we cannot guarantee that the *CS* will entirely precede the corresponding *RS*. However, we can impose an additional requirement, based on j', so that $i < j' - l_1$, with i being the index that was initially inputted to *BMD*. As a result the segment $Update_{k...i}$ will precede the $Current_{j...m}$, and will not be overwritten when the reconstruction of

the former is conducted.

Ensuring that these relationships hold, allows the reconstruction to be performed in-place, as all detected *CSs* will not precede the respective *RSs* and will have not been overwritten when they need to be copied. Of course these additional requirements reduce the number of the available *CSs* for each *RS*, resulting in larger delta scripts, compared to the out-of-place reconstruction which preserves the current firmware throughout the reconstruction process.

4.4 Orientation of Dfinder to Executable Files

Although the above methodology works remarkably well for unstructured binary files, it fails to utilise some traits of executable files, such as firmware images. For instance, in actual firmware images, and especially when linker optimisations are used, large common segments are placed in the same order in the two firmware versions. Hence, we decided to modify the algorithm to utilise these traits and achieve higher compression ration on executable files.

To this end, we introduced three additional instructions, which enable a more efficient delta encoding. More specifically, when using *Dfinder* on actual firmware images, we realized that the majority of the instructions were *Type 1* or *Type 3 COPY* instructions, while for most of them the distance of the initial address (m) of the *CS* (in the current firmware) from the one of the respective *RS* (in the updated firmware) was less than 256. For such cases, the initial address of the *CS* could be expressed using a single byte (instead of three), as the distance of it from the initial address of the *RS*. Besides, an mentioned above many *CSs* and *RSs* that were encoded using *Type 1 Copy* instructions, had identical initial addresses; meaning that these two segments originate at the same address in the two firmware images.

To exploit these traits, we introduce the following instructions: *Copy_ type1_ rel*, *Copy_ type3_ rel* and *Copy_ in-place* (see Figure 4.9 and Figure 4.10). The first two instructions resemble *Type 1* or *Type 3 COPYs*, which represent the initial address of the *CS* using a single byte, as the distance from the initial address of the reconstructed *RS*, with total overhead of $\gamma = 3$ bytes. The latter instruction does not have a *CS* initial address field at all, as the two matching segments have the same initial address in the two firmware versions. Subsequently, *Copy_ in-place* instructions have total overhead of $\delta = 2$ bytes. Besides, we modified *Optimiser* to also take into account these additional instructions, achieving optimal delta encoding.

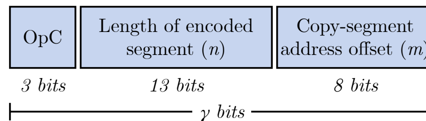


Figure 4.9: *Copy_ type1_ rel* and *Copy_ type3_ rel* instructions format

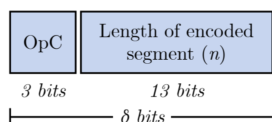


Figure 4.10: *Copy_ in-place* instructions format

Furthermore, due to micro-changes in the updated firmware, caused by function or variable shifts, often entire regions are identical between the two versions, except for a few bytes; even a single byte. This results in delta scripts which contain many *ADD* instructions, which have a single byte as payload. We refer to these *ADDs* as *light ADDs*. To avoid the overhead of *light ADDs* in the final delta script we introduce a processing step, using the following approach. Once the delta script has been computed, all instructions are iterated over, *light ADDs* are tracked, and their single-byte payload is stored in a byte-array. Moreover, all *light ADDs* are removed and we construct a bit-array, which has length equal to the number of the remaining instructions in the delta script. This bit-array is used for recording the existence of a removed *light ADD* before each such instruction. Moreover, these two arrays are augmented to the delta script. Hence, during the reconstruction, before executing each instruction, the receiver checks the corresponding entry of the bit-array and if the value is 1 (meaning that there was initially a *light ADD* before the current instruction), the respective single-byte payload can be retrieved using the byte-array. Using this approach, we can avoid the additional overhead of *light ADDs*, resulting in substantially smaller delta scripts.

Chapter 5

OTAP Testbed

In this chapter, we describe the architecture of the testbed we implemented to evaluate the performance of *Dfinder* in an actual OTAP scenario for resource-constrained IoT networks. Specifically, we designed the testbed to support devices running the Contiki-NG OS, while the firmware upgrade of these devices takes place using the delta script that is calculated using *Dfinder*. The system also enables the transmission of the entire updated firmware image; however, as we primarily focus on the support of incremental programming, we omit some implementation details and mainly discuss the components that allow devices to be upgraded using delta scripts, securely and seamlessly. The architecture consists of three entities: the OTAP server, the IoT network, and the border router. The OTAP server is an interconnected computing system, which is needed for the cross-compilation of the updated firmware image, the computation of the delta script, and its transmission to the devices of the IoT network. In order to build a small IoT network, we used the Zolertia RE-Mote platform, running the Contiki-NG OS. Finally, the border router facilitates the interconnection of the IoT network with the rest of the internet, and subsequently the end-to-end communication of the IoT devices with the OTAP server.

5.1 IoT Technologies and Standards

The main IoT technologies and standards that are utilised in our system are discussed in this section. We present some network protocols which belong to different layers of an IoT protocol stack, such as the one presented in Figure 5.1. It must be noted that the figure is limited to present the protocols (and a few more) that were used in this thesis; however, a typical IoT stack will usually include more protocols, some of which are not limited to the IoT ecosystem.

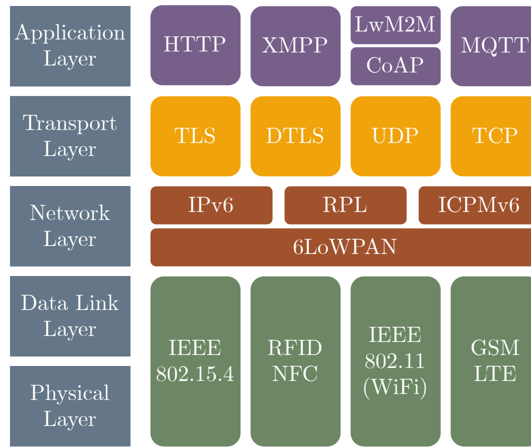


Figure 5.1: An IoT protocol stack

5.1.1 IEEE 802.15.4

IEEE 802.15.4 [50] (RFC 4944) is one of the most established standards that enable the communication in Low-power and Lossy Networks (LLNs). It is a packet-based radio protocol that defines the physical layer (PHY) and the medium access control (MAC) specifications, serving as the foundation for several protocol stacks (e.g. Zigbee, WirelessHART, SmartLink, etc.). The first version of the standard was announced by IEEE 802.15 Task Group 4 (TG4) in 2003 and since then many versions of it have emerged (e.g. IEEE 802.15.4e, IEEE 802.15.4g).

The standard supports multiple modulation schemes, including BPSK, ASK, O-QPSK, MR-FSK, MR-OFDM, and MR-O-QPSK, and its operational frequency includes the 2.4GHz industrial, scientific and medical (ISM) band to facilitate worldwide availability, while 868 and 915 MHz bands are also supported for Europe and USA, respectively. Besides, the standard has maximum data rate of 250 Kbps, depending on the used modulation scheme and operational frequency band. Furthermore, the standard can provide fully acknowledged frame delivery, which may be desirable in environments with high interference. In the original specification, the frame size (MTU) is limited to 127 bytes, aiming at reducing the probability of frame errors by forcing a small frame size. Finally, it applies collision avoidance through CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) on the physical layer. This way, the devices have to sense the channel before starting the transmission and are allowed to proceed only if the channel is idle.

5.1.2 6LoWPAN

6LoWPAN [66] (RFC 6282) is an acronym of IPv6 over Low-Power Wireless Personal Area Networks and defines the standards for IPv6 communication over the IEEE 802.15.4 protocol. It is a network stack (or adaptation layer) that allows IPv6 (RFC 2460) packets to be carried efficiently within constrained links, such as

IEEE 802.15.4 links, as IPv6 needs an MTU of at least 1280 bytes [1]. This way, the end-to-end native communication of IoT devices with any IPv6 network is enabled.

This is achieved by compressing the IPv6 headers and fragmenting the IPv6 datagrams sourced from the internet, by an adaptation layer placed below IP, for supporting the IEEE 802.15.4 maximum MTU. On the other hand, the headers need to be decompressed and the IPv6 datagrams reassembled when the traffic is forwarded to the internet, for supporting the IPv6 minimum MTU. Besides, 6LoWPAN enables stateless auto-configuration, as IoT devices inside a 6LoWPAN network can automatically generate their own IPv6 addresses.

5.1.3 RPL

RPL [2] [35] (RFC 6550) is a routing protocol that enables IPv6 datagrams to be routed in constrained IoT networks. It is a distance-vector protocol [3] that computes a Destination Oriented Directed Acyclic Graph (DODAG), based on a set of metrics and constraints. The computed DODAG has a single root node (DODAG route) and exists a path from every node to this root. Finally, this graph is automatically built by the devices, exchanging ICMPv6 control packets.

5.1.4 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) [14] (RFC 7252) was standardized by IETF as a lightweight alternative to HTTP, for enabling the Machine-to-Machine (M2M) communication of resource-constrained devices. It follows the client-server model. where clients can make requests to servers and the servers send back responses. Similar to HTTP, CoAP is a RESTful protocol, featuring the typical GET, POST, PUT and DELETE methods, which can be used for accessing the resources of a server, using a URI path. In addition, using the Resource Discovery mechanism, servers can provide a list of their accessible resources, along with metadata about them. A client may "subscribe" for one or more of these resources using the *Observe* option, so that it receives updates about state changes from the server, when such changes occur to the corresponding resources. CoAP is designed to have minimal RAM requirements (~10 KiB) and small code space. Besides, the protocol uses UDP links, and the communication can be secured using the Datagram Transport Layer Security (DTLS) protocol [64] (RFC 4347) using Pre-shared Key (PSK), Raw Public Key (RPK), or X.509 certificates, while it can also offer reliable communication, acknowledging the successfully received messages.

---

[1] It must be noted that the frame format of IEEE 802.15.4g does not have this limitation.

[2] Pronounced "ripple"

[3] Distance-vector protocols are more suitable for constrained devices, compared to the link-state counterparts, as the former need to store substantially less control plane data at each device.

5.1.5 OMA Lightweight Machine to Machine (LwM2M)

The Lightweight Machine to Machine (LwM2M) protocol [92] is a lightweight device management protocol, located at the application layer of the OSI stack, and is specified by the Open Mobile Alliance (OMA). It focuses on reducing of energy and data consumption needed for the M2M communication, rendering it ideal for low-power and low-capacity IoT devices. LwM2M follows the client-server communication model (see Figure 5.2) and implements it extending the CoAP, so that LwM2M management operations and responses are mapped to respective CoAP counterparts (Coap method calls and response codes). This communication scheme requires an LwM2M server and an LwM2M client. The client is usually located in an IoT network, while the server can be an interconnected computing system, typically deployed in the cloud or a data center, and can simultaneously serve hundreds of IoT networks. The clients have to subscribe to one or more LwM2M servers, offering access to their resources. During their communication, both the server and the client can make requests. Hence, these two entities must not be confused with the typical client-server communication model where only the client makes requests and the server responds. In the case of LwM2M, the server acts as the coordinator of the communication. In addition, LwM2M defines a simple resource model, where the client has different objects, which are definitions of some of its resources (e.g. temperature reading from a sensor). Besides, the client can have several instances of the same object. By interacting with these resources the server can change several aspects in the device during run time, like the firmware that the device should run.

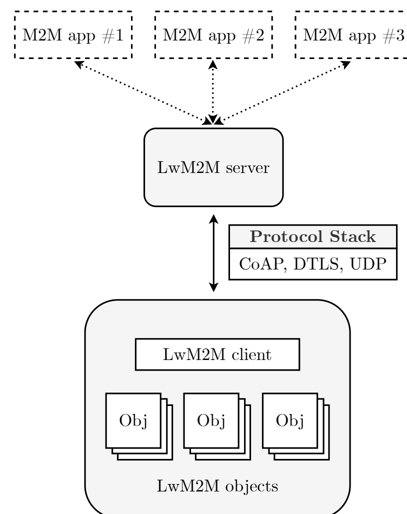


Figure 5.2: LwM2M architecture

5.1.6 Eclipse Leshan

Eclipse Leshan [4] is an OMA LwM2M client and server implementation in Java, which provides a highly documented API, enabling developers to create their LwM2M applications. Finally, it is based on Californium CoAP and Scandium DTLS implementations and supports IPSO objects.

5.1.7 Contiki-NG

Contiki-NG is an open-source, event-driven operating system, developed for resource-constrained IoT devices. It has been designed to run on devices with limited processing power and memory, and has been ported on a wide range of platforms, rendering it an ideal solution for developing novel IoT applications. Contiki-NG includes a plethora of easily extendable general libraries for sensing and actuating, as well as some cryptography libraries and mechanisms for estimating power consumption. Besides, it contains multiple demo applications which can be easily tested on target devices, or using the Contiki's network simulator, called Cooja.

In order to enable the communication between nodes, Contiki-NG uses the traditional OSI (Open Systems Interconnection) stack for implementing its network protocol stack, called *NETSTACK*. It includes traditional TCP/IP protocols (e.g. UDP, TCP IPv4, IPv6, HTTP, ICMP), as well as some recently standardized low-power wireless standards, such as RPL, CoAP, LwM2M, and 6LoWPAN. On the networking layer, it relies on an IPv6 stack, offering lightweight implementations for IP, UDP, and TCP. The networking layer is composed of two sub-layers, the upper IPv6 layer and the lower adaptation layer (6LoWPAN), which both operate on top of the IEEE 802.15.4. Besides, regarding routing, Contiki-NG applies RPL in the network topology. Contiki-NG supports different MAC protocols, such as CSMA/CA (non-beacon enabled) and TSCH [5]. The former is a protocol where the radio is always in listening mode, and is allowed to transmit only once it has checked the medium and verified that it is not occupied. Hence, the power consumption is high, as the radio is constantly turned on. The latter is a MAC layer that uses channel hopping and allows nodes to turn off the radio for fixed periods of time. Additionally, Contiki-NG implements lightweight stack-less threads, called protothreads, and all of them share the same memory stack. By not using separate stack spaces, these threads are non-preemptive, avoiding the context switching when another thread is yield.

5.1.8 RPL Border Router

The border router or edge router is a device found at the edge of an IoT network, enabling the communication of the latter with external networks. As mentioned above, an adaptation layer, in the form of 6LoWPAN, is required for enabling the

[4] `www.eclipse.org/leshan`

[5] Contiki-NG does not support ContikiMAC Radio Duty Cycling Protocol.

communication of IoT devices with the internet over IPv6. Contiki-NG implements the 6LoWPAN adaptation layer using an RPL border router, which acts as a 6LoWPAN router. These devices are tasked with routing the ingress and egress traffic towards the destination, as well as handling the compression/decompression of IPv6 headers and fragmentation/defragmentation of IPv6 datagrams, as specified by 6LoWPAN. Finally, the border router will also record the addresses of the IoT devices, so that they are accessible in the internet as shown in Figure 5.3.

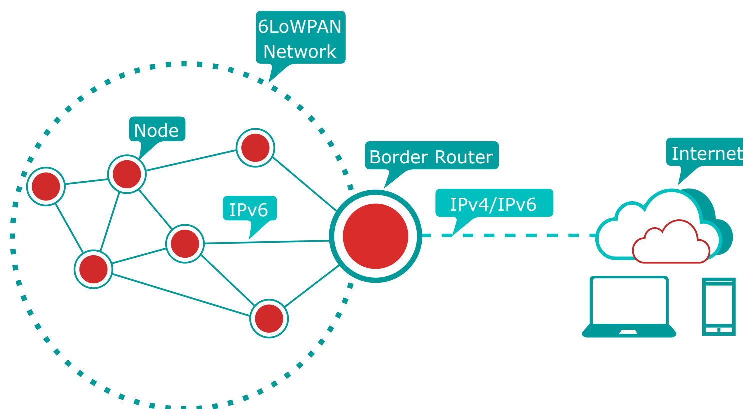


Figure 5.3: An interconnected IoT network, using a border router[6]

5.1.9 Zolertia RE-Mote Platform

In order to construct an IoT network of the testbed, we used the Zolertia RE-Mote platform (see Figure 5.4). Besides, we used a single RE-Mote in the role of the 6LoWPAN border router, enabling the interconnection of the IoT devices with our OTAP server (a laptop connected via USB with the border router). The platform is equiped with the Zolertia Zoul module. Zoul is based on the Texas Instruments CC2538, which has a 32-bit ARM Cortex-M3 system on chip (SoC) (clocked up to 32 MHz), with integrated IEEE 802.15.4-compliant tranceiver, at 24 GHz. Besides, it features 512 KiB of program flash and 32 KiB of RAM, while it also offers hardware acceleration for a wide range of cryptography functions (AES-128/256, SHA2, ECC-128/256). Additionally, it also includes a Texas Instruments CC1200 868/915 MHz RF transceiver, for enabling dual-band communication. The on-board Texas Instruments CC1200 enables multiple modulation formats (2-FSK, 2-GFSK, 4-FSK, 4-GFSK, MSK, OOK). These traits, along with their small power consumption, have empower the support of the Zolertia RE-Mote platform from the Contiki-NG community.

[6]Source: `www.zolertia.io/6lowpan-iot-protocol`

Figure 5.4: Zolertia RE-Mote Platform[7]

5.2 Testbed Implementation

In this section we present the components of the testbed, along with their main functionality.

5.2.1 OTAP Server

The OTAP server is an interconnected computing system that is responsible for preparing the delta script using the *Dfinder* algorithm and transmitting it to the devices of an IoT network, allowing their upgrade by reconstructing the updated firmware. In particular, this communication is enabled, using the LwM2M protocol, where the OTAP server corresponds to the Eclipse Leshan server and the IoT device is an LwM2M client that registers to the server. Concurrently, the Eclipse Leshan provides a graphical interface to manage the firmware update procedure, such as the one shown in Figure 5.5 that presents the IoT devices that have registered to the server.

Each time a new firmware version has been released, the update sequence begins. First, the OTAP server collects the newly compiled binary file and uses it, in conjunction with the firmware image that the target devices currently run, to calculate the delta script (using *Dfinder*). Then, some additional information (manifest) is augmented to the delta script, composing the final OTAP image, which is transmitted to the devices using LwM2M. The structure of the OTAP image consists of the manifest, the manifest signature, and the delta script itself. The manifest describes the corresponding firmware images (current and updated ones), containing valuable information about them (e.g. their digest), while at the same time provides information needed for the successful reconstruction of the updated firmware image. The manifest signature is important for identifying the authenticity of the delta script. In the following paragraphs, we describe in detail

[7]Source: `www.zolertia.io/product/re-mote`

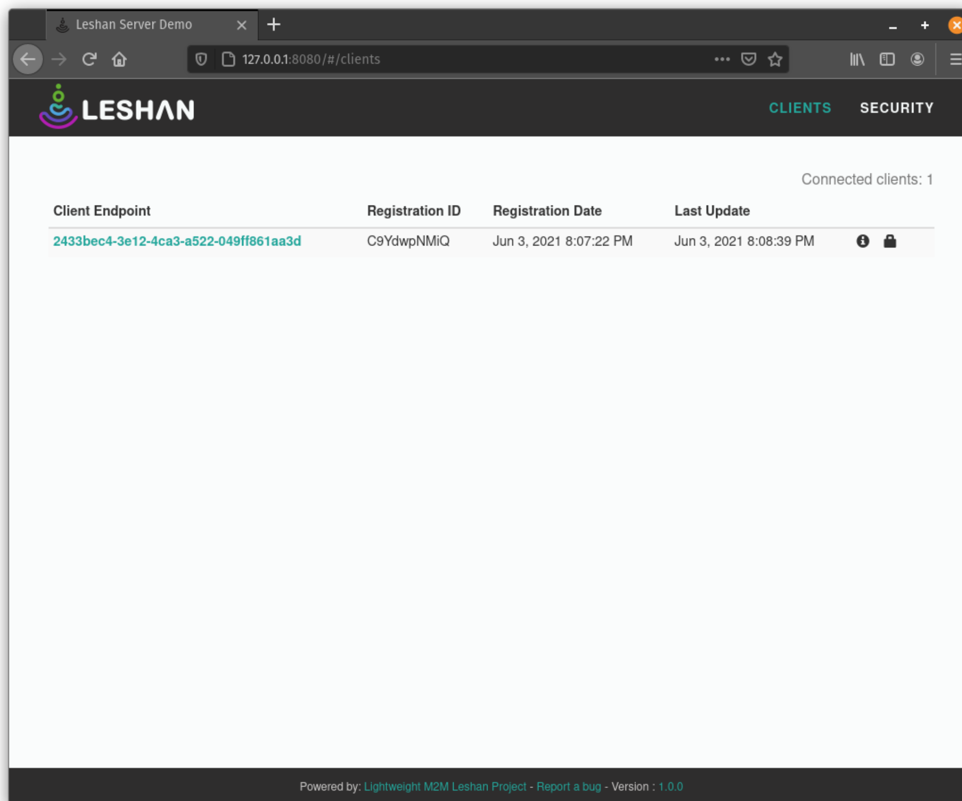


Figure 5.5: List of IoT devices registered to the Leshan server

how the different parts of the above structure are produced and how they enhance the system's operation.

Manifest generation The manifest consists of the fields described in Table 5.1. Among others, the manifest specifies if the OTAP server is ready to transmit the entire image of the updated firmware version or a delta script. If it is the latter case, the manifest must also specify if the reconstruction has to be performed *in-place* or *out-of-place*. Upon receipt, the OTAP client on the device checks and confirms the manifest fields and proceeds with the update process.

Manifest version ID	In case multiple versions of manifest exists each one has an ID which describes the structure of the manifest.
Monotonic Sequence Number (MSN)	The UTC timestamp at the time of manifest generation, it is used to declare the freshness of the new image.
Vendor ID	Describes the vendor of the device in which the image is compatible.

Class ID	Describes the class of the device in which the image is compatible.
Precursor image digest	Contains the SHA-256 digest of the precursor image and is used so that the device can assure that it bears the appropriate image to apply the delta script.
Expiration time	The date that the installed firmware expires and a new one has to be installed.
Update type	Used for identifying that the update is incremental (based on delta scripts) or not.
Reconstruction type	Declares if the reconstruction has to be performed *in-place* or not.
Storage location	Defines the flash address where the delta script should be stored at (e.g. memory offset).
Payload digest	The SHA-256 hash of the payload used to verify the integrity of the received delta script.
Payload size	The size of the payload used to confirm that the application partition has enough space to store the received delta script.
Reconstruction checksum	Contains the CRC-16 value of the updated firmware.
Reconstruction location	Defines the flash address where the updated firmware will be reconstructed at.

Table 5.1: Manifest structure

Signature generation The OTAP server applies a digital signature to the manifest using the user's private key (also known as signing key) to confirm the image author and guarantee the integrity of the delta script. Note that the public-private key pair is generated using any of the Contiki-NG supported elliptic curves (e.g. secp256r1).

Delta script generation Once the updated firmware version has been compiled, the resulted firmware image is collected, in conjunction with the current firmware image. These two files have Intel-Hex format and need to be converted to binary files. Then, *Dfinder* is used for computing the delta script based on the two images.

5.2.2 OTAP Client

OTAP client is based on the LwM2M client and it is installed on IoT devices. OTAP client ensures efficient firmware over the air updates of the devices while enabling effective remote management of the device. Those two client's characteristics are the cornerstone of the over-the-air updates, as will be discussed below.

5.2.3 Device Preparation

Updating the firmware of an IoT device requires an initial preparation of the device. For this reason, a base image is created which has to be installed on the devices (via USB) before their deployment, to enable future OTAP support. Specifically, the flash is divided unequally into four partitions. The first partition contains the firmware installer, the second partition contains the device metadata, the third partition contains the running application image, while the fourth partition remains empty until the first incremental update occurs. The last partition is used for storing the received delta scripts, and if the reconstruction is out-of-place, for storing the reconstructed updated firmware. On the other hand, if the reconstruction is performed in-place, the fourth partition will only store the delta script, as the reconstructed image will replace the current application image, located at the third partition.

- **Firmware Installer:** The firmware installer is a stand-alone Contiki-NG application, that features the necessary functionality for supporting LwM2M communication, and allowing the reconstruction of the updated firmware, according to the received delta script. Each time a new firmware update is available, the device is set to OTAP mode and boots to the firmware installer.

  The firmware installer is designed to connect to the OTAP server by the OTAP client and securely download and verify the OTAP image. Besides, it stores the received delta script at the end of the fourth partition, allowing the reconstruction and installation of the updated firmware.

- **Metadata:** The metadata partition consumes one page (2048 Bytes) of the program flash and contains information about the installed application image. The metadata fields are the DTLS credentials, the public part of the verification key, the Monotonic Sequence Number (MSN), the vendor and the class IDs of the device, the digest, the expiration time, and the storage location. The metadata are important for a variety of reasons. Particularly, the DTLS credentials are used by the device to communicate with the OTAP server over the DTLS protocol. The verification key is used to verify the integrity and authenticity of the received delta script, while the monotonic sequence number ensures its freshness. The vendor and class IDs describe the characteristics and the family of the device. The image digest is the SHA-256 hash of the installed image and is essential in differential updates, as it provides guarantee that the reconstruction will be performed based on the correct firmware versions. The expiration time defines the date that the installed image ceases to be supported and a new one has to be installed. Finally, the storage location describes the flash address where the current image is installed.

- **Application Image:** this partition accommodates the installed application image. The application image is the firmware which the device loads after

each boot if no update request occurs. As the first application image, we use a simple application that blinks the device on-board LED every 1 second and connects to the OTAP server by the OTAP client. The application image is updated upon OTAP server request.

5.2.4 Delta Script Download and Firmware Installation

Each time a new firmware version is available, the OTAP server informs the target devices, using the proper LwM2M resource, to stop the execution of the current firmware and boot to the firmware installer. This operation can be done very easily through the Leshan platform, selecting the target device from the list of the registered devices, and pressing the "Factory Reset" button (see Figure 5.6).

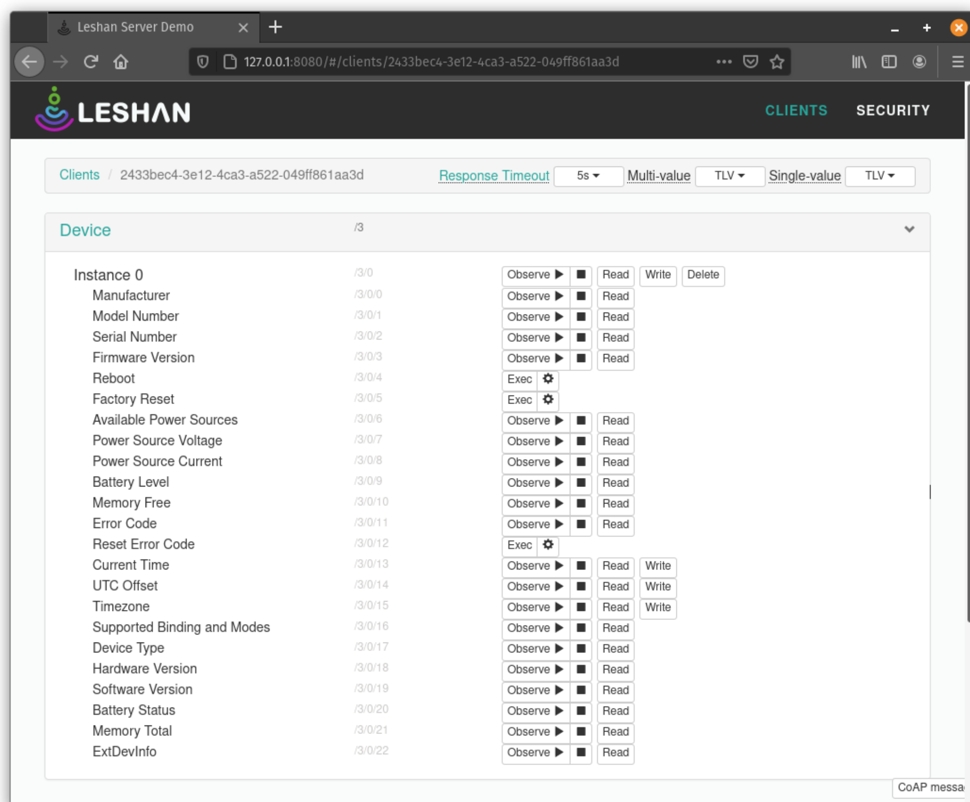


Figure 5.6: The available resources and methods offered by an LwM2M client, when the device runs the firmware image

Once the selected device has booted the firmware installer, the OTAP server initiates the transmission of the manifest. Again, this operation can be done through the Leshan interface (see Figure 5.7), by pressing the "Write" button and selecting the computed OTAP image. The received manifest is stored in the RAM of the

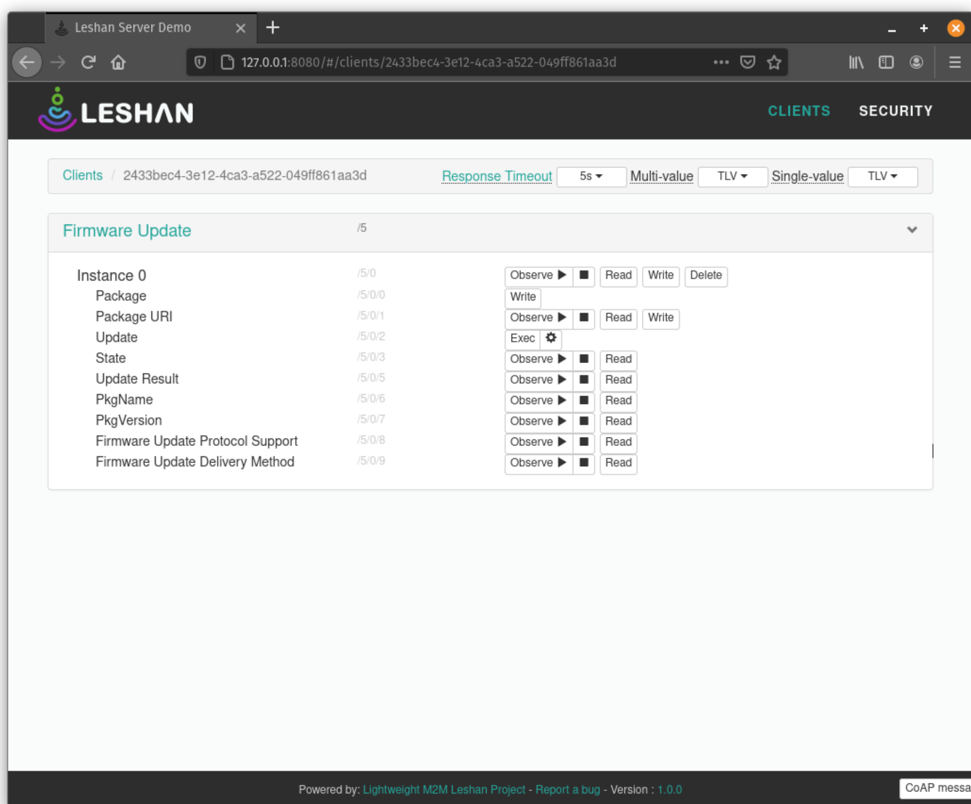


Figure 5.7: The available resources and methods offered by an LwM2M client, when the device runs the firmware installer

device and based on the contained signature, as well as the public key stored in the metadata section, the firmware installer validates the integrity and the authenticity of the manifest. If the signature is valid the next step is to identify the manifest fields to prevent the installation of an unsuitable (e.g. wrong class, vendor ID) or out-of-order firmware (e.g older MSN). The identification of the manifest fields results from their comparison with the corresponding metadata fields. In case the digital signature or the manifest fields are invalid, then the device aborts the update process and boots to the application partition where the current firmware image is already installed. On the other hand, if the manifest is successfully verified, the firmware installer continues the download of the delta script while the SHA-256 digest is calculated on it. Once the downloading is completed, the firmware installer checks if the calculated digest of the received data is the same as the payload digest field of the manifest. If it is not the case, the update is aborted, otherwise, the firmware installer originates the reconstruction of the updated firmware.

Using the corresponding field (Reconstruction type) of the manifest, the firmware installer can tell if the reconstruction of the updated firmware has to be performed

in-place or not. If it is the former case, it determines the address of the currently installed firmware (based on the device metadata) and starts the reconstruction at that address. However, if the reconstruction is performed out-of-place, the updated firmware is reconstructed at an address specified in the received manifest, within the fourth partition (as long as it does not overlap with the delta script). In either case, the CRC-16 (cyclic redundancy check) of the reconstructed image is computed and compared with the one presented in the manifest. If these values are equal, the reconstruction was successful, the corresponding fields at the metadata partition are updated respectively, and the device boots to this just reconstructed firmware image. On the other hand, if the computed CRC-16 differs from the one present in the manifest, the reconstruction (and thus the update) has failed. If this is the case, we must study two instances. First, if the reconstruction was done in-place, the firmware image which was installed in the third partition before the update will have been overwritten by the falsely reconstructed image, and thus been corrupted. In this case, the firmware image is unusable and the device needs to boot to the firmware installer and remain inoperational, waiting for a new OTAP attempt. However, if the reconstruction was done out-of-place, the current version will have not been overwritten and the device can simply boot to this version, staying operational until its upgrade is reattempted. The entire flow of the delta update process is depicted in Figure 5.8.

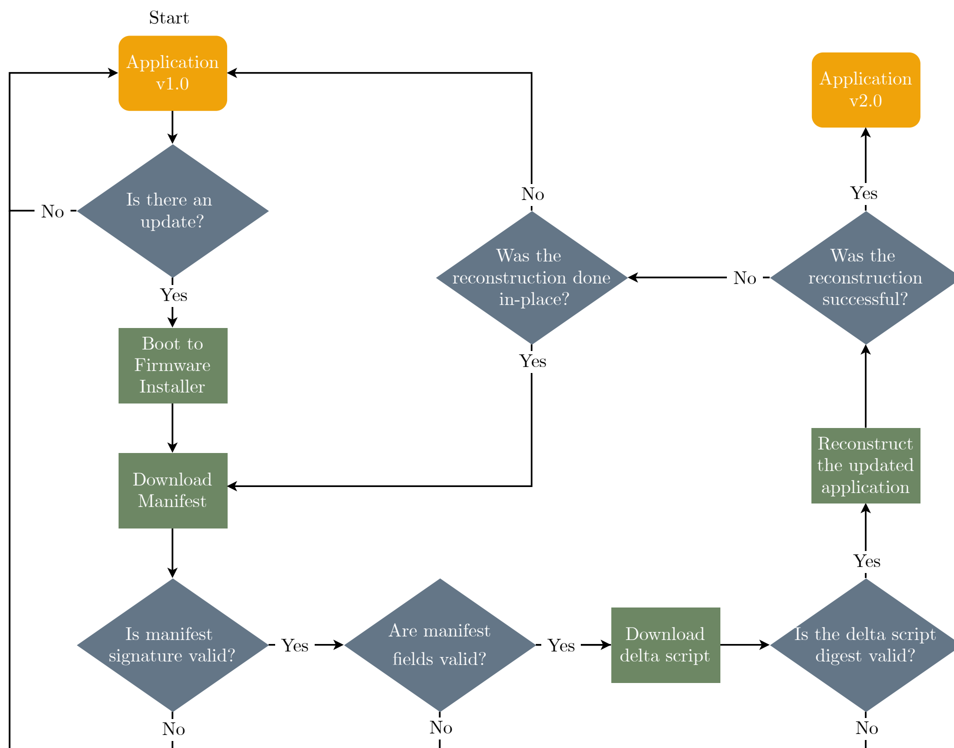


Figure 5.8: Flow of firmware upgrade from v1.0 to v2.0 using delta image

5.2.5 Firmware Reconstruction

In order to reconstruct the updated firmware successfully, the instructions in the delta script have to be interpreted by the device in order. Interpreting these instructions, data are written to sequential memory regions, progressively reconstructing the updated firmware. However, in order to be able to write data to a flash address, the corresponding block (of 2048 Bytes) has to be erased entirely first. As a result, the aforementioned interpretation of the delta script would not work, as any flash write would require the data that have been written to that block previously to be erased.

For this reason, we used a buffer of 2048 Bytes that we progressively fill by interpreting the instructions, and only when it is full, we write the contained data to the corresponding flash block. This method works exceptionally when the reconstruction is out-of-place, as long as we have previously erased the entire flash region where the updated firmware will be located after the reconstruction. However, when the reconstruction is done in-place, we are not able to erase the entire reconstruction flash region, because this is the flash region where the current firmware located. As a result, all copy instructions would be rendered false. In order to avoid this, each block has to be erased separately before it is overwritten with the data contained in the buffer. This approach also affects the reconstruction time (and energy consumption) as a mass-erase is more efficient than deleting individual memory blocks.

Chapter 6

Experimental Evaluation

In this chapter, we will focus on the evaluation of the incremental programming approach that we proposed in this thesis. To this end, we compare *Dfinder* with other differencing algorithms (see Table 6.1), regarding the execution time, the memory footprint, and the resulted delta script size. In order to enable their accurate comparison, we benchmark them inputting both randomly generated binary files, as well as actual images of different Contiki-NG firmware versions. This way, we can measure their performance under various changes that may occur between two firmware versions with higher accuracy. Finally, we measure how the dissemination performance and the energy consumption in the network are affected, when the update is done incrementally, using the proposed strategy (based on the delta scripts that *Dfinder* computes).

BSDiff
Delta Generator
JojoDiff
R3diff
Rdiff
RMTD
Xdelta3
Dfinder

Table 6.1: The benchmarked differencing algorithms

Apart from *JojoDiff* [1], all other algorithms that were used during the evaluation have already been discussed in Chapter 2. *JojoDiff* is a differencing utility implemented in C++, which has been developed to produce delta scripts for executable files. The utility tries to find the differences between two executable files, using a heuristic algorithm, which has constant space and linear time complexity. Hence,

[1] `www.jojodiff.sourceforge.net`

it may not be able to find the smallest delta script for most firmware updates, but it is fast and requires a fixed amount of memory. Besides, it is an open-source project which enables its seamless modification and integration to OTAP systems.

Regarding *RMTD* and *R3diff* (the differencing algorithm used in *R3* [19]), we accessed them using the publicly available repository that the authors of *R3* have provided, and we executed them using the parameters: $\alpha = 2$ and $\beta = 5$. Hence, we run *R3diff* as a component of an OTAP system, called *R3*, which also includes a similarity preserving mechanism, called *R3sim*. *BSDiff*, *Xdelta3* and *Rdiff* are well-known difference utilities, which can easily be accessed using the package manager of most Linux distributions. Moreover, we configured *Xdelta3* to avoid additional compression on the output, as we are interested in the differencing capabilities of the utility. Furthermore, due to copyright issues, we could not access the source code of *Delta Generator* (*DG*); however, the authors kindly provided us a Windows 10 executable. Hence, as *DG* had to be run on a separate computing system, the measured memory consumption and time performance cannot be compared with the ones of the other utilities. Finally, all experiments (except the ones that involved *DG*) were conducted on a Linux system, which includes an Intel® Xeon® CPU E5-2630 v2 clocked at 2.60GHz, with 4 GiBs of RAM and 5.4.0-70-generic kernel.

6.1 Evaluation using Randomly Generated Binary Files

Although *Dfinder* is oriented to executable files, such as firmware images, it is useful to examine how it performs on unstructured binary files. The motive is that when building an updated firmware image, changes in the source code may result in large segments of code that are not common between the two versions. As a result, some difference utilities may be unable to detect common patterns in these regions, resulting in large delta scripts. For instance, we were able to catch on this effect when multiple *printf* calls were added in the updated source code of a Contiki-NG application.

In order to conduct the experiments, we generated a thousand random binary files of various sizes, which were coupled into pairs of two, with the first representing the old file and the other one the new (updated) file. Then, we inputted these pairs to the differencing algorithms to compute the corresponding delta scripts, also tracking the memory consumption and execution time. In addition, in order to measure the similarity of the binary files of each input pair, we used the *Levenshtein distance* -or *edit distance*- metric, which computes the number of operations that need to be performed on a sequence, to transform it into another one. Thus, this metric can be used for expressing how unsimilar two files -or two sequences- are. However, due to the recursive nature of the algorithm, it fails on large inputs and the execution time is extravagant. Thus, we introduce a naive estimation of the *Levenshtein distance* of two files, which requires fewer memory resources and time, and operates by computing the *Levenshtein distance* metric on smaller blocks of

these files. First, the two files are divided into equal-sized blocks. As the two files may have unequal sizes, the first (old) file has n such blocks, while the latter (new) has m blocks. Then, the estimation of the *Levenshtein distance* is calculated using Equation 6.1. Note that $f_{NEW:i}$ denotes the ith block of the new file and $f_{OLD:j}$ the jth block of the old file, respectively.

$$\begin{aligned} Levenshtein_distance_EST(f_{OLD}, f_{NEW}) = \\ \sum_{i=1}^{m} \min_{j=1}^{n} Levenshtein_distance(f_{OLD:j}, f_{NEW:i}) \end{aligned} \tag{6.1}$$

Besides, the proposed estimation takes into account the size of the new file, so that the measured *Levenshtein distance* does not exceed it. Hence, this metric is more appropriate, as the produced encoding shall enable the reconstruction of the new file version and not the transformation of the old version into the new one, as the original *Levenshtein distance* does. On the other hand, the original *Levenshtein distance* measures the operation needed for transforming a sequence into another; hence, it measured metric may be way larger than the size of the updated file. Additionally, we measured the execution time of each algorithm using the Linux time utility, and the memory consumption was measured using the *Valgrind* suite [2] (see Listing 6.1).

```
$valgrind --tool=massif --pages-as-heap=no --massif-out-file=massif\
command; grep mem_heap_B massif | sed -e 's/mem_heap_B=\(.*\)/\1/'\
| sort -g | tail -n 1
```

Listing 6.1: The use of Valgrind for obtaining the peak memory consumption of a bash command

Figure 6.1 presents the size of the delta script that the benchmarked differencing algorithms computed. The x-axis represents the logarithm (with base 10) of the *Levenshtein distance* of each input pair of binary files, while the y-axis represents the corresponding delta script size in Bytes. Moreover, the figure does not contain any information about *JojoDiff*, *Rdiff* and *DG*, as these algorithms consistently produced patches of size equal to the one of the updated (new) file. Moreover, we did not use the *Dfinder* version which is oriented to executable files (see section 4.4), as the generic one is more appropriate for unstructured files, which do not contain specific patterns. Besides, we inferred that both *Dfinder* versions (the one that enables the in-place reconstruction and the one that enables the out-of-place reconstruction of the updated file) perform almost identically; hence we aggregate these two *Dfinder* versions in the plot.

Based on the plot, we infer that as the unsimilarity of input files increases, all algorithms tend to compute larger delta scripts. However, *BSDiff* and *Xdelta3* generally produce the smallest ones, while *Dfinder* tends to create similar-sized delta scripts to the ones that *RMTD* computes. Moreover, compared to other algorithms, *BSDiff* seems to be less prone to the increase of unsimilarity of input files, as its incline is not that steep, compared to the rest of the algorithms. This trait can be explained by the fact that *BSDiff* uses compression on the computed

[2] valgrind.org

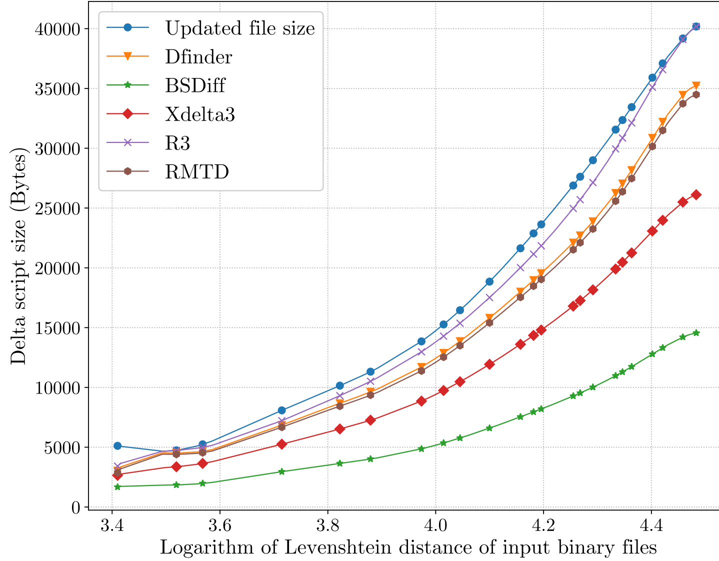


Figure 6.1: The size of the resulted delta script size versus the unsimilarity (expressed by the Levenshtein distance) of the two input binary files

delta script, causing its further shrinkage. Based on the above, we expect *BSDiff* to perform better than the other algorithms when the firmware modifications are extended, as it will be able to encode these large uncommon segments more efficiently, in terms of size.

Figure 6.2 depicts the execution time of each differencing algorithm, and how it is affected by the unsimilarity of the input files. First, we can notice that as the unsimilarity increases, *Dfinder* tends to have larger execution time, while *Rdiff*, *BSDiff*, and *Xdelta3* have constant execution time. The unpredictable execution time of *R3diff* is due to the application of *R3sim* (the similarity preserving mechanism of *R3*) on the inputs. In contrast to previous studies, we were able to run *RMTD* for relatively large files ($\sim$ 30 KiB), on a system with mediocre memory resources. However, the execution time was extravagant ($\sim$ 100 seconds) and hence we did not include these measurements in the plot as they would disturb the final result.

Figure 6.3 showcases how the execution time of the algorithms is affected by the total size of the two input files. Based on it, we can infer that the size of the input files does not provide a good factor for determining the execution time of *Dfinder*. Additionally, this Figure allows us to observe that *R3diff* needs more time as the size of input files gets larger. Besides, in this plot we can observe, that *R3diff* takes longer than *JojoDiff* to compute delta scripts; this information was not showcased in the previous plot, as these higher values of execution time for

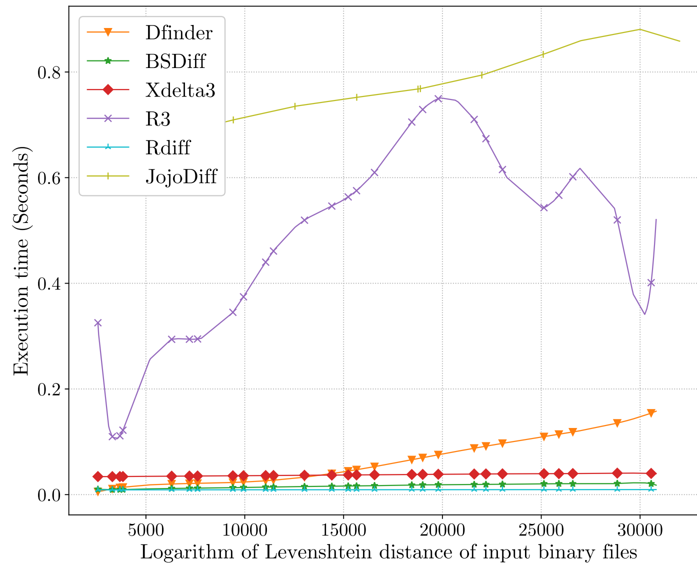


Figure 6.2: Execution time of differencing algorithms versus the unsimilarity (expressed by the Levenshtein distance) of the two input binary files

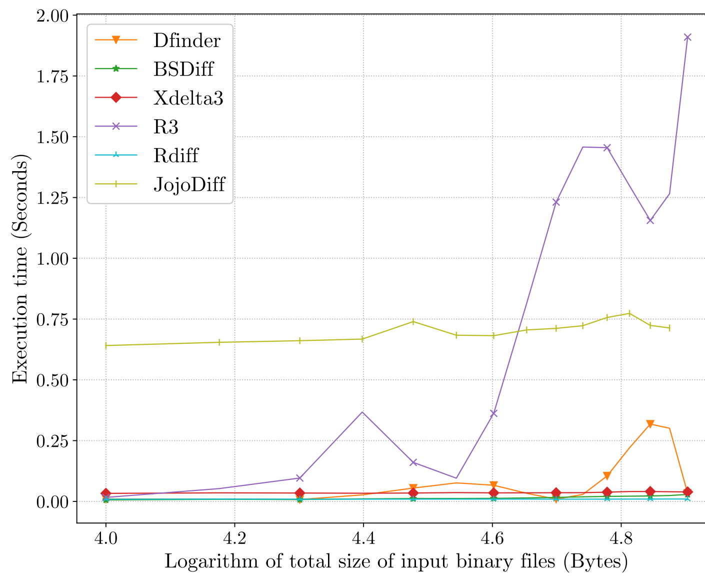


Figure 6.3: Execution time of differencing algorithms versus the total size of the two input binary files

R3diff were presented as outliers in the measured results and were hidden in the plot due to the application of polynomial fitting.

Although the execution time of a differencing algorithm is a major factor for determining its performance, memory consumption is far more important, especially when the algorithm is used by a firmware server that simultaneously serves hundreds of networks. Figure 6.4 presents the peak memory consumption of the algorithms we benchmarked. In contrast to the execution time, the memory consumption is primarily affected by the size of the input files, as they use auxiliary data structures with size proportional to that of the input files. For instance, *Dfinder* uses the byte-array T and other arrays, that have size of $2 * (length(f_{NEW}) + length(f_{OLD}))$ bytes. Based on our experiments, we infer that *Rdiff* has the lowest memory consumption, which is to be expected as it is a block-level differencing utility and its operation does not require any sophisticated data structures. On the other hand, *RMTD* is the most memory-intensive algorithm, which is due to the two-dimensional array that it uses for the detection of the common segments. We can also observe that *Xdelta3*, *Rdiff*, *JojoDiff*, and *BSDiff* have constant memory requirements (for the tested inputs). Finally, *Dfinder* and *R3diff* have almost identical memory consumption, which is significantly less than that of the other algorithms (apart from *Rdiff*).

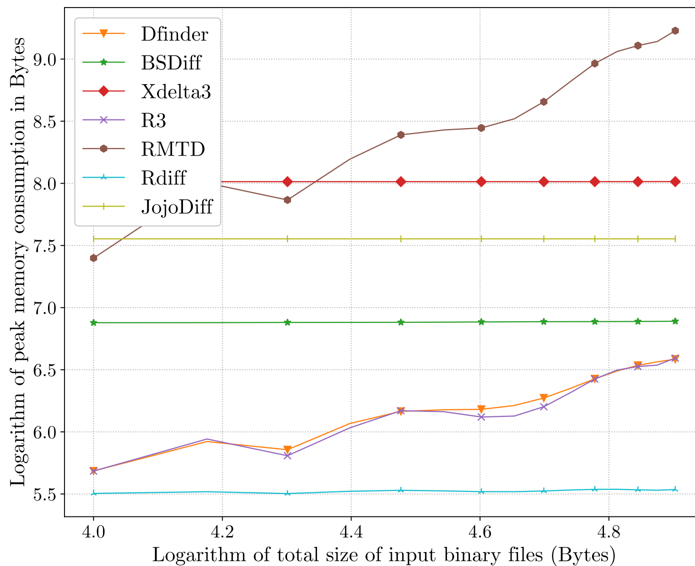


Figure 6.4: Peak memory consumption of differencing algorithms versus the accumulated size of the two input binary files

6.2 Evaluation using Actual Firmware Images

In order to benchmark the differencing algoritms on actual Contiki-NG firmware images, we used the *hello-word* application, which is included in Contiki's repository. Introducing incremental modifications in the source code, we were able to produce seven firmware versions, as shown in Table 6.2.

Version 1	Base version
Version 2	Two initialized variables were added, and their values are printed using an additional *printf* call
Version 3	A new function was implemented and called
Version 4	An additional instruction is added in the implementation of the new function
Version 5	No modifications (same as version 4)
Version 6	An additional *printf* call was added
Version 7	Three new functions were implemented and called

Table 6.2: The different firmware versions that we inputted to the differencing algorithms

During the compilation of a firmware image, the compiler may use various optimisation techniques, which can affect its final size and subsequently the performance of differencing algorithms (compression ratio). For instance, redundant or unused code may be removed, reducing the overall firmware image size. Additionally, some functions may be inlined by the compiler, favoring the execution time over the size of the firmware. As a result, references to such (inlined) functions are replaced by the corresponding code; hence, their relocation will not cause the typical function shift effects. As the proposed technique is not limited to a specific platform, we want to suppress any optimisations imposed by the compiler, as they may vary across platforms. This way, we are able to better observe how the differences between two firmware versions affect the resulted delta script size. To this end, we compiled the firmware image of each produced version, using both the default optimisations (`-Os`), as well as suppressing the optimisations, using the flags: `OPTIMIZATIONS=-O0` and `SMALL=0`, as specified in Contiki-NG wiki [3]. It must be noted that this way, not all optimisations are suppressed; for instance, unused functions may still be removed during the compilation. Hence, once we had compiled the images for all seven firmware versions, we applied each differencing algorithm on each pair of sequential firmware versions, creating six delta scripts that allow the transition from one version to the next. During this process, we monitor the delta script size, as well as the execution time and memory consumption of each algorithm. Besides, this process was done for both optimised and unoptimised code, and the experiments were repeated 100 times to compute the execution time of

[3] `www.github.com/contiki-ng/contiki-ng/wiki/Platform-openmote-cc2538`

each algorithm more accurately.

6.2.1 Optimised Code

First, table 6.3 depicts the size of the firmware images, that were compiled using the default optimisations. It must be noted that V_4 and V_5 firmware images are identical as no modifications were introduced to V_5. Besides, V_3 image is identical to V_4 -and V_5-, due to linker optimisations. In the following tables, with bold are the best values achieved by the differencing algorithms, for each firmware update.

Firmware version	Size in Bytes
V_1	41608
V_2	41628
V_3	41628
V_4	41628
V_5	41628
V_6	41740
V_7	41740

Table 6.3: The size of the compiled firmware images using the default optimisations

In Table 6.4 is depicted the delta script size that the differencing algorithms computed. Based on it, we can obtain several insights. First, due to the larger delta scripts that all algorithms produced, we can infer that large volume of differences exist between the V_1 and V_2 images, as well as the V_5 and V_6 images. These two firmware updates ($V_1 \rightarrow V_2$ and $V_5 \rightarrow V_6$) share the addition of a *printf* call in the source code of the updated firmware version. Besides, we observe that *Rdiff* performs poorly during these two firmware updates, as the resulted delta script is similar-sized to the image of the updated firmware version. Additionally, during these two firmware updates, the out-of-place-reconstruction *Dfinder* version and *BSDiff* compute the smallest delta scripts, among all other algorithms.

Firmware Update	RMTD	DG	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	3771	5910	8286	2392	41639	5465	**1057**	3568	3939
$V_2 \rightarrow V_3$	13	**11**	31	31	2065	16	160	63	15
$V_3 \rightarrow V_4$	5	**2**	26	26	9	5	140	53	-
$V_4 \rightarrow V_5$	5	**2**	26	26	9	5	140	53	-
$V_5 \rightarrow V_6$	3871	6026	8362	2479	41751	5569	**1175**	3719	4043
$V_6 \rightarrow V_7$	**46**	63	71	57	2065	54	209	92	60

Table 6.4: Delta script size (in bytes) produced by the benchmarked differencing algorithms for optimised code

During the other firmware updates, all algorithms can compute relatively small delta scripts, with *RMTD* and *DG* computing the smallest ones. Besides, both *Dfinder* versions have remarkably good results as they compute delta scripts only 25 bytes larger than the smallest one. In addition, *JojoDiff* failed to produce delta scripts when inputted two identical files. Finally, *BSDiff* produced the largest delta scripts when the updated firmware version introduces a small volume of differences.

Firmware Update	RMTD	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	151.480	0.072	0.033	**0.020**	0.483	0.023	0.049	1.807
$V_2 \rightarrow V_3$	154.524	0.039	0.039	**0.015**	7.345	0.016	0.050	1.167
$V_3 \rightarrow V_4$	153.790	0.052	0.037	0.014	8.800	0.015	bf0.011	-
$V_4 \rightarrow V_5$	161.040	0.042	0.039	0.016	10.412	0.018	**0.008**	-
$V_5 \rightarrow V_6$	133.800	0.045	0.022	**0.019**	0.292	0.023	0.029	1.532
$V_6 \rightarrow V_7$	104.939	0.021	0.023	**0.011**	4.358	**0.011**	0.033	0.779

Table 6.5: Mean execution time (in seconds) of differencing algorithms for optimised code

In Table 6.5 we can see the mean execution time of each algorithm during the experiments. We can observe that regardless of the firmware update, *RMTD* takes the longest time to compute a delta script, while *Rdiff* requires the least time. In addition, *BSDiff* is the second-fastest algorithm, with *Xdelta3* and both *Dfinder* versions taking the third place. Next in order comes *JojoDiff* and then *R3diff*. Interestingly, *R3diff* requires more time to produce delta scripts for firmware updates with small differences.

Firmware Update	RMTD	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	1735978794	4168820	4167398	**125022**	3926883	7777418	103265334	7777498
$V_2 \rightarrow V_3$	1736005122	4162861	4166957	**39006**	3921774	7777498	103265334	7777498
$V_3 \rightarrow V_4$	1736005082	4162858	4166954	**32862**	3921763	7777498	68531254	-
$V_4 \rightarrow V_5$	1736005082	4162858	4166954	**32862**	3921763	7777498	68531254	-
$V_5 \rightarrow V_6$	1745357186	4177690	4176288	**125022**	3935951	7777722	103265446	7777498
$V_6 \rightarrow V_7$	1745342858	4174096	4178175	**39006**	3932340	7778170	103265446	7777498

Table 6.6: Peak memory consumption (in bytes) of differencing algorithms for optimised code

In Table 6.6 we can observe the peak memory consumption of the differencing algorithms. We infer that *Rdiff* is the most space-efficient algorithm, while *R3diff* and *R3diff* come next. In addition, *BSDiff* and *JojoDiff* consume almost double the memory. Finally, *RMTD* and *Xdelta3* are quite memory-intensive algorithms for all firmware updates.

6.2.2 Unoptimised Code

Linker optimisations may not be applicable for all processor families. Besides, the effects of some firmware modifications may be hidden by the application of optimisations. Hence, we repeated the above experiments producing firmware images suppressing the linker optimisations. In Table 6.7 is shown the size of these compiled firmware images. We can observe that these firmware images are quite larger that the linker-optimised ones from the previous experiments. Additionally, mind that only the V_4 and V_5 firmware images are identical, as not modifications have been introduced to the source code updated version.

Firmware version	Size in Bytes
V_1	73472
V_2	73504
V_3	73620
V_4	73644
V_5	73644
V_6	73756
V_7	73924

Table 6.7: The size of the compiled firmware images without linker optimisations

Firmware Update	RMTD	DG	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	4101	6491	11759	2663	73518	5993	**1429**	4002	4360
$V_2 \rightarrow V_3$	4291	6541	12531	3023	73634	6143	**1523**	4194	4480
$V_3 \rightarrow V_4$	4054	6453	11783	2566	73658	5954	**1296**	3946	4251
$V_4 \rightarrow V_5$	5	**2**	33	33	11	5	142	53	-
$V_5 \rightarrow V_6$	4127	6559	11660	2600	73770	6016	**1369**	4002	4378
$V_6 \rightarrow V_7$	4277	6629	12270	2910	73938	6170	**1513**	4187	4567

Table 6.8: Delta script size (in bytes) produced by the benchmarked differencing algorithms for unoptimised code

Table 6.8 shows the delta script size that the algorithms produced for the firmware updates. Compared to the previous experiments (using linker-optimised firmware images), we can infer that all algorithms produce quite larger delta scripts. This implies that even small modifications in the firmware source code can result in large differences and subsequently large delta scripts. Hence, we can infer the same insight as we did for optimised firmware images when the updates included a large volume of differences. As a result, *BSDiff* produces the smallest delta scripts, while *Dfinder* produces the second smallest ones.

In Table 6.9 and Table 6.10 is shown the execution time and the peak memory consumption of the differencing algorithms, respectively. We can also obtain

Firmware Update	RMTD	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	298.538	0.053	0.035	**0.012**	0.724	0.017	0.030	0.830
$V_2 \rightarrow V_3$	286.104	0.052	0.035	**0.012**	0.722	0.017	0.027	0.803
$V_3 \rightarrow V_4$	285.835	0.050	0.033	**0.011**	0.730	0.017	0.028	0.821
$V_4 \rightarrow V_5$	292.823	0.038	0.042	0.011	16.09	0.014	**0.006**	-
$V_5 \rightarrow V_6$	287.351	0.053	0.041	**0.012**	0.728	0.016	0.030	0.813
$V_6 \rightarrow V_7$	287.721	0.054	0.031	**0.015**	0.731	0.016	0.027	0.835

Table 6.9: Mean execution time (in seconds) of differencing algorithms for unoptimised code

Firmware Update	RMTD	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	5409050762	7356671	7354695	**125948**	6923551	7968626	103428282	7777498
$V_2 \rightarrow V_3$	5426112262	7366969	7364339	**125948**	6933177	7968986	103428398	7777498
$V_3 \rightarrow V_4$	5429648850	7370913	7368856	**125948**	6936808	7969498	103428422	7777498
$V_4 \rightarrow V_5$	5429633194	7364462	7368558	**33788**	6931267	7969594	68563270	-
$V_5 \rightarrow V_6$	5446165162	7379666	7377820	**125948**	6945902	7969818	103428534	7777498
$V_6 \rightarrow V_7$	5470975066	7395673	7393328	**125984**	6960896	7970602	103428702	7777498

Table 6.10: Peak memory consumption (in bytes) of differencing algorithms for unoptimised code

the same insights as we did for optimised code; however, the execution time and memory consumption may be affected by the larger size of the compiled firmware images. Interestingly, the memory consumption of *BSDiff* and *JojoDiff* is not affected by the larger size of input files, but they still consume more memory than *Dfinder*. Additionally, the memory consumption of *BSDiff*, *Xdelta3* and *JojoDiff* is not affected by the larger input files. Regarding the execution time, only the one of *RMTD*, *JojoDiff*, and *R3diff* seem to be significantly affected. On the contrary, the execution time of *Dfinder* is not affected by the larger size of input firmware images, but requires more memory for the used data structures.

6.3 Energy Consumption and Dissemination Performance

In this section, we examine if the reduction of transmitted data actually affects the update time and the energy consumption in the network and the magnitude of it. To this end, we used the testbed we presented in Chapter 5 and implemented a simple Contiki-NG project that includes the LwM2M module, enabling the communication of the devices with the firmware server. Besides, introducing the modifications that are presented in Table 6.2, we produced seven firmware versions and compiled the corresponding images. As the RE-Motes are based on CC2538, we compiled those images, so that the board operates at PM1 low power mode, using CSMA/CA MAC protocol. As a result, the antenna is always turned during the delta/firmware transmission, but it is turned off during firmware reconstruction.

More specifically, we used a laptop in the role of the firmware server, which is

connected over USB with a RE-Mote, which runs the RPL border router Contiki-NG application. Besides, another RE-Mote is located three meters away from the border router, which acts as the target device that we want to update. Additionally, the laptop is able to communicate with the target device, via the border router, using Contiki's tunslip6 utility. Tunslip6 creates a virtual network interface (tun) and uses SLIP (serial line internet protocol) to encapsulate and pass IP traffic over the two ends of the serial line. Using this setup, we update the target RE-Mote, both incrementally, transmitting the delta script that *Dfinder* computed, and transmitting the entire firmware image. To measure the time needed to update the device, we used the Contiki-NG *Rtimer* library, which allows high clock resolution and can calculate time intervals in ticks. Thus, for incremental updates, we were able to track the time needed to download the delta script, and reconstruct the updated firmware; otherwise, we track the time needed to download the entire firmware.

In order to compute the energy consumption, we use Contiki's *Energest* module, which operates by tracking the time various hardware components are turned on. By knowing the current these components operate, we are able to compute the energy consumption. The supported energest types that can be monitored in all Contiki-NG platforms are shown in Table 6.11. The first two types track the different modes of CPU, while the next two track when the radio is on and the respective state. Besides, to provide a more accurate measurement of the energy consumption, we use Rtimer library to track the duration of flash erases and writes. Hence, we are able to track how long the CPU is in low power mode or not, how long the antenna is listening or transmitting, and how long the device performs flash erases and writes. Besides, we can compute the energy consumption of each state/operation using Equation 6.2 and the total energy consumption can be calculated as the sum of the individual energy consumptions for each state/operation.

Type	Purpose
ENERGEST_TYPE_CPU	The CPU is active
ENERGEST_TYPE_LPM	The CPU is in low power mode
ENERGEST_TYPE_TRANSMIT	The radio is transmitting
ENERGEST_TYPE_LISTEN	The radio is listening

Table 6.11: The predefined energest types

$$\text{Energy consumption [mJ]} = \frac{\text{Energest_Value} * \text{Current} * \text{Voltage}}{\text{RTIMER_SECOND}} \tag{6.2}$$

The voltage, current and RTIMER_SECONDS can usually be found in the datasheet of the used platform. For instance, for CC2538 SoC [4], the used voltage is 3V,

[4] www.ti.com/lit/ds/symlink/cc2538.pdf?ts=1623877461875&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FCC2538

RTIMER_SECONDS is 32768 and the current per operation is shown in Table 6.12.

State/Operation	Current
CPU	13mA
LPM	0.6 mA
RX (0 dBm)	24mA
TX (-100 dBm)	24mA
Flash erase	13 mA
Flash write	8 mA

Table 6.12: Electric current per operation/state of CC2538 SoC

For consistency, apart from *Dfinder* we used the other differencing algorithms too, to observe how they compare to each other regarding the delta scripts produced for each firmware update. It must be noted that the firmware image for each version was $\sim$ 81 KiBs. Based on the larger delta scripts by all differencing algorithms, we can infer that the firmware updates with large volume of differences are the following: $V_1 \rightarrow V_2$, $V_5 \rightarrow V_6$, and $V_6 \rightarrow V_7$.

Firmware Update	RMTD	DG	Dfinder (in-place)	Dfinder (out-of-place)	Rdiff	R3diff	BSDiff	Xdelta3	JojoDiff
$V_1 \rightarrow V_2$	7668	12069	19184	4809	79656	11093	1803	7327	7863
$V_2 \rightarrow V_3$	11	11	38	38	2065	14	154	62	13
$V_3 \rightarrow V_4$	5	2	35	35	11	5	142	53	-
$V_4 \rightarrow V_5$	5	2	35	35	11	5	142	53	-
$V_5 \rightarrow V_6$	7684	12141	19134	4788	79708	11115	1797	7352	7885
$V_6 \rightarrow V_7$	7619	11968	19104	4700	77859	11048	1705	7272	7802

Table 6.13: The delta script size (in bytes) produced by the benchmarked differencing algorithms for an LwM2M-enabled Contiki-NG application

Regarding the update time (see Figure 6.5), we can infer that the overall time needed for updating the target device, is substantially reduced when it is updated incrementally, due to the vast reduction of the transmitted data. We can also infer that the updates with a large volume of differences require more time, due to the larger delta script size. However, even in these cases, the update time is substantially improved, compared to transmitting the entire firmware.

Furthermore, due to the reduction of the transmitted data, the antenna needs to stay active (in listening mode) for a shorter period of time, as it is deactivated when the reconstruction begins. Hence, in Figure 6.6 we can observe the energy consumed by the node when it is updated incrementally is reduced up to 96% for updates with a small volume of differences and 92% for updates with a large volume of differences, when the reconstruction is done out-of-place.

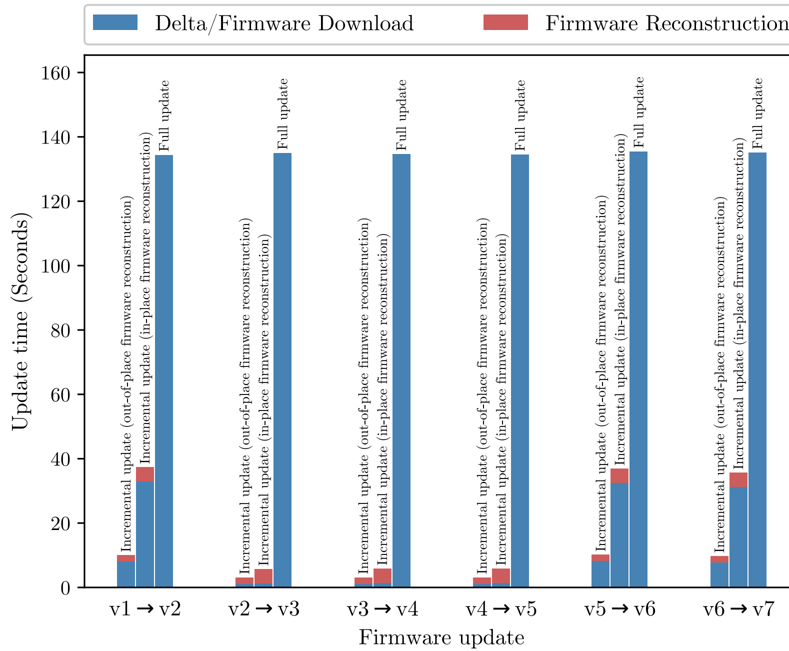


Figure 6.5: Time in seconds needed for updating the target IoT device

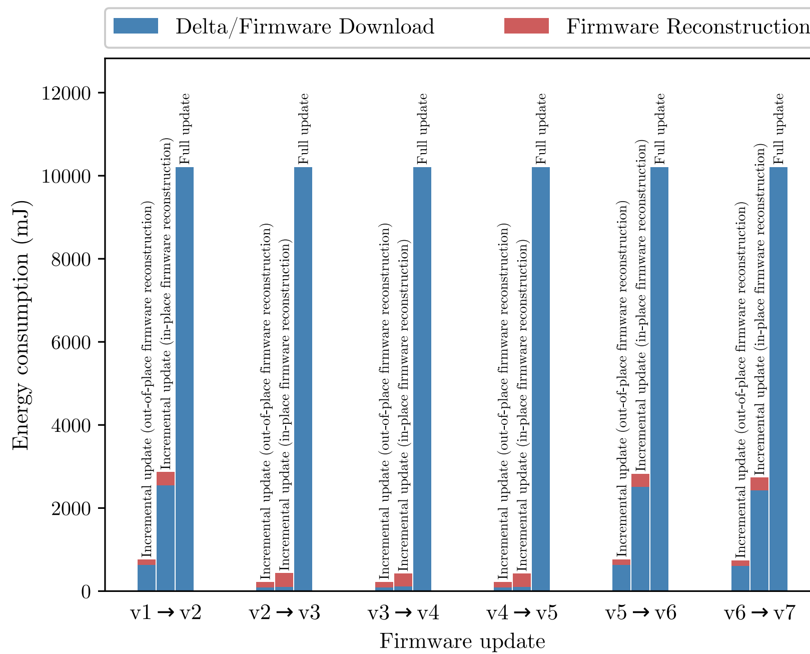


Figure 6.6: Energy consumption during firmware update

Additionally, in Figure 6.7 is shown the energy consumed for each stage/operation during the update. Interestingly, RX is the most energy-intensive state, as the antenna needs to stay active in listening mode, waiting to receive more data. We can also infer that the energy consumed for flash operations when the reconstruction of the updated firmware is done in-place is more compared to the out-of-place reconstruction. The reason is that in the first case multiple erases must be performed for separate flash blocks, while in the second case only one mass-erase is needed for a large flash region, which is more efficient.

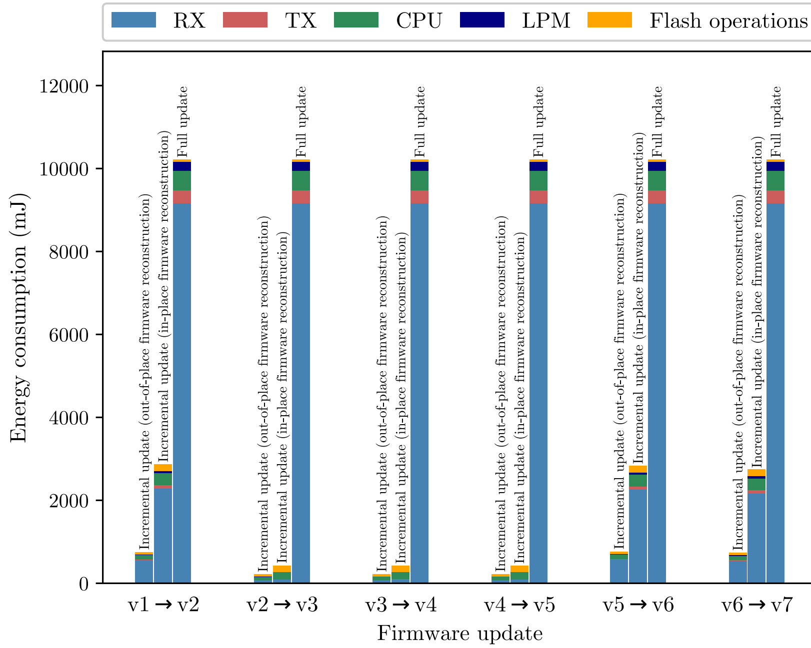


Figure 6.7: Energy consumed by the target IoT device per operation/state during firmware update

Summarizing, if the update includes a small volume of differences between the two sequential firmware versions, both *Dfinder* versions operate exceptionally, reducing the time and energy consumption to 96% for out-of-place reconstruction and 94% for in-place. On the other hand, when there is a large volume of differences, the out-of-place *Dfinder* version retains its benefits, reducing the time and energy consumption up to 90%, while the in-place version achieves a 70% reduction.

Chapter 7

Conclusion and Future Work

In this thesis, we focused on OTAP support in resource-constrained IoT networks. We underlined the primal limitations such networks face and presented a technique called incremental programming, which enables their efficient upgrade by transmitting an encoded patch (called delta script) that describes the differences between the two firmware versions. Besides, we presented the related literature that mainly consists of works that introduce differencing algorithms, that aim at producing the smallest delta scripts possible, keeping the execution time and memory consumption low. Furthermore, some network protocols have been introduced that allow the efficient dissemination of the delta script or the entire firmware in the network, while also ensuring its authenticity and freshness, securing the upgrade process.

Additionally, we developed and presented a byte-level differencing algorithm, called *Dfinder*, that utilises enhanced suffix arrays, resulting in the computation of small delta scripts, with small execution time and memory footprint. Besides, the algorithm can construct delta scripts that enable the reconstruction of the updated firmware image in-place, starting at the flash address where the current firmware image originates. This trait renders it an ideal solution for incremental OTAP solutions that focus on platforms with very constrained storage resources. In order to evaluate its performance, we developed a testbed that enables the OTAP of an IoT network, transmitting a delta script that a firmware server remotely computes, using *Dfinder*.

During the evaluation, we compared *Dfinder* with other differencing algorithms, regarding the produced delta script size, the execution time, and the memory consumption. Based on our experiments, we found out that our algorithm consistently produces very small delta scripts when small modifications have been introduced to the firmware source code, while it also produces the second smallest delta scripts when these modifications are extended. Besides, *Dfinder* has quite a small execution time (less than 0.05 seconds) and the memory consumption is one of the smallest ones, among the algorithms we tested. The last trait renders it ideal for firmware servers that may simultaneously serve hundreds of IoT networks. Finally, using the testbed that we implemented, we showcased that updating IoT networks

incrementally (based on the delta scripts computed by *Dfinder*) can reduce both the update time and energy consumption in the network up to 96%.

As future work, we plan to port our solution to other operating systems, such as Zephyr RTOS, and other platforms. Besides, we want to extend our testbed and evaluate it in a heterogeneous multihop network, which consists of many different IoT boards and OSes. This way, we could find out the limitations that these platforms may face (e.g.lack of support for a communication protocol) and try to address these issues, offering workarounds (e.g. implementations using alternative protocols). Besides, we plan to modify the upgrade process, so that the delta script instructions can be bundled into distinct LwM2M messages and avoid the storage of the entire delta script in the flash, before the reconstruction stage, rendering our solution more space-efficient. Finally, we want to study why some modifications in *Contiki-NG* (e.g. the insertion of a *printf* call) result in substantially larger delta scripts than other modifications do. This way, we could try to mitigate these effects and produce even smaller delta scripts for Contiki-NG platform.

Chapter 8

Acknowledgments

This research has been 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).

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86, 2004. The 9th International Symposium on String Processing and Information Retrieval.

[2] Alberto Apostolico. The Myriad Virtues of Subword Trees. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, pages 85–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[3] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis. Firmware over-the-air programming techniques for iot networks – a survey, 2020. Accessed: 2020-10-02.

[4] Jonathan Payne Arthur van Hoff. Generic Diff Format Specification, 1997.

[5] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghu, and G. Tsudik. Assured: Architecture for secure software update of realistic embedded devices. *IEEE Transactions On Computer-aided Design Of Integrated Circuits and Systems*, pages 2290–2300, 2018.

[6] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Hermann Foot, Florian Grieskamp, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and et al. sacabench.

[7] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Hermann Foot, Florian Grieskamp, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. SACA Bench. April 2019. Publisher: Technische Universität Dortmund.

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

[9] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on*

Discrete Algorithms, SODA '97, page 360–369, USA, 1997. Society for Industrial and Applied Mathematics.

[10] F. Bodon and L. Rónyai. Trie: An alternative data structure for data mining algorithms. *Mathematical and Computer Modelling*, 38:739–751, 2003.

[11] S. Brown and C. Sreenan. Software Updating in Wireless Sensor Networks: A Survey and Lacunae. *Journal of Sensor and Actuator Networks*, pages 717–760, 2013.

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

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

[14] Louis Coetzee, Dawid Oosthuizen, and Buhle Mkhize. An analysis of coap as transport in an internet of things environment. 05 2018.

[15] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12, 2008.

[16] C. Dong and F. Yu. An efficient network reprogramming protocol for wireless sensor networks. *Computer Communications*, 55, 2014.

[17] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao. R2: Incremental Reprogramming Using Relocatable Code in Networked Embedded Systems. *IEEE Transactions on Computers*, 62:1837–1849, 2013.

[18] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen. Elon: enabling efficient and long-term reprogramming for wireless sensor networks. In *Proc. of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '10*, page 49. ACM Press, 2010.

[19] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *Proc. of the IEEE INFOCOM*, pages 315–319, 2013.

[20] F. D'Souza and D. Panchal. Advanced encryption standard (aes) security enhancement using hybrid approach. In *Proceedings of the International Conference on Computing, Communication and Automation (ICCCA)*, pages 647–652, 2017.

[21] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proc. of the 4th international*

conference on Embedded networked sensor systems - SenSys '06, page 15. ACM Press, 2006.

[22] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 2004.

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

[24] M. Ersue, D. Romascanu, J. Schoenwaelder, and A. Sehgal. Management of Networks with Constrained Devices: Use Cases, rfc7548. Technical report, 2015.

[25] M. Farooq and T. Kunz. Operating Systems for Wireless Sensor Networks: A Survey. *Sensors*, pages 5900–5930, 2011.

[26] J. Fischer and Florian Kurpicz. Dismantling divsufsort. *ArXiv*, abs/1710.01896, 2017.

[27] Andres Gomez. On-demand communication with the batteryless mirocard: Demo abstract. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 629–630, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Roberto Grossi. Suffix Trees and their Applications in String Algorithms. 1997.

[29] A. Hagedorn, D. Starobinski, and A. Trachtenberg. 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 2008)*, pages 457–466, 2008.

[30] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. SOS -A Dynamic operating system for Sensor Networks. In *Proc. of MobiSys*, 2005.

[31] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38:2162–2178, 01 2009.

[32] J. Hu, Chun Jason Xue, Yi He, and Edwin H.-M. Sha. Reprogramming with Minimal Transferred Data on Wireless Sensor Network. In *Proc. of the 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 160–167, Macau, China, 2009. IEEE.

[33] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: An empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7(2):192–214, April 1998.

[34] S. Hyun, P. Ning, A. Liu, and W. Du. Seluge: Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 445–456, 2008.

[35] Oana Iova, Pietro Picco, Timofei Istomin, and Csaba Kiraly. Rpl: The routing standard for the internet of things... or is it? *IEEE Communications Magazine*, 54:16–22, 12 2016.

[36] Jaein J. and D. Culler. Incremental network programming for wireless sensors. In *Proc. of the First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks.*, pages 25–33, 2004.

[37] J. Jeong. Node-level Representation and System Support for Network Programming, 2003.

[38] O. Kachman. Configurable Reprogramming Scheme for Over-theAir Updates in Networked Embedded Systems, 2016.

[39] O. Kachman. Effective multiplatform firmware update process for embedded low-power devices, 2018.

[40] O. Kachman and M. Balaz. 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)*, pages 1–4. IEEE, 2016.

[41] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP'03, page 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

[42] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006.

[43] Toru Kasai, Gunho Lee, Hiroki Arimura, Arikawa Setsuo, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching*, volume 2089, pages 181–192, 2001.

[44] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In Ricardo Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching*, pages 186–199, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[45] A. Kishore. Turning internet of things (iot) into internet of vulnerabilities (iov) : Iot botnets, 2017.

[46] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005. Combinatorial Pattern Matching (CPM) Special Issue.

[47] D. Korn, J. MacDonald, J. Mogul, and K. Vo. *The VCDIFF Generic Differencing and Compression Data Format.* RFC Editor, 2002. Published: RFC 3284.

[48] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proc. of the Second European Workshop on Wireless Sensor Networks, 2005.*, pages 354–365, Istanbul, Turkey, 2005. IEEE.

[49] J. Koshy and R. Pandey. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys 2005 - Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pages 243–254, 2005.

[50] Anis Koubaa, Mário Alves, and Eduardo Tovar. Ieee 802.15.4 for wireless sensor networks: A technical overview. 01 2005.

[51] S. Kulkarni and L. Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 7–16, 2005.

[52] Juha Kärkkäinen, Giovanni Manzini, and Simon Puglisi. Permuted longest-common-prefix array. pages 181–192, 06 2009.

[53] P. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: Secure Dissemination of Code Updates in Sensor Networks. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, 2006.

[54] K. Lehniger and S. Weidling. The Impact of Diverse Execution Strategies on Incremental Code Updates for Wireless Sensor Networks. In *Proceedings of the 8th International Conference on Sensor Networks*, pages 30–39. SCITEPRESS - Science and Technology Publications, 2019.

[55] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[56] P. Levis, N. Patel, D. Culler, and S. Shenker. 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.* USENIX Association, 2004.

[57] T. Liu, C. Sadler, P Zhang, and M. Martonosi. 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*, page 256. ACM Press, 2004.

[58] J. Macdonald. Xdelta - open-source binary diff. 2011.

[59] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[60] Michael A. Maniscalco and S. J. Puglisi. Faster lightweight suffix array construction.

[61] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM J. Exp. Algorithmics*, 12, June 2008.

[62] Giovanni Manzini. Two space saving tricks for linear time lcp array computation. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004*, pages 372–383, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

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

[64] Nagendra Modadugu and Eric Rescorla. The design and implementation of datagram tls. 12 2003.

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

[66] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, page 78–82, New York, NY, USA, 2007. Association for Computing Machinery.

[67] D. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exp.*, 27:983–993, 1997.

[68] Guiqiang Ni, Yingzi Yan, J. Jiang, Jianmin Mei, Zhilong Chen, and Junxian Long. Research on incremental updating. In *ICC 2016*, 2016.

[69] S. Nisha and M. Farik. RSA Public Key Cryptography Algorithm – A Review. *International Journal of Scientific & Technology Research*, pages 187–191, 2017.

[70] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202, 2009.

[71] R. Panta and S. Bagchi. Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks. In *IEEE INFOCOM 2009 - The 28th Conference on Computer Communications*, pages 639–647. IEEE, 2009.

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

[73] Colin Percival. Naive differences of executable code, 2003.

[74] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys*, 2019.

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

[76] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4–es, July 2007.

[77] Sanguthevar Rajasekaran and Marius Nicolae. An elegant algorithm for the construction of suffix arrays. *Journal of Discrete Algorithms*, 27:21 – 28, 2014.

[78] N. Reijers and K. Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *Proc. of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 60–67. Association for Computing Machinery, 2003.

[79] M. Sanvido, F.R. Chu, A. Kulkarni, and R. Selinger. nand Flash Memory and Its Role in Storage Architectures. *Proc. of the IEEE*, 96:1864–1874, 2008.

[80] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

[81] Martin Senft. Suffix tree based data compression. In *Proceedings of the 31st International Conference on Theory and Practice of Computer Science*, SOFSEM'05, page 350–359, Berlin, Heidelberg, 2005. Springer-Verlag.

[82] N. Shafi, K. Ali, and H. Hassanein. No-reboot and zero-flash over-the-air programming for Wireless Sensor Networks. In *Proc. of the 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 371–379, 2012.

[83] J. Shi, J. Wan, H. Yan, and H. Suo. A survey of Cyber-Physical Systems. In *Proc. of the International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2011.

[84] A. Shoufan and N. Huber. A fast hash tree generator for Merkle signature scheme. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 3945–3948, 2010.

[85] Anish Man Singh Shrestha, Martin C. Frith, and Paul Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in Bioinformatics*, 15(2):138–154, 2014. _eprint: https://academic.oup.com/bib/article-pdf/15/2/138/562332/bbt081.pdf.

[86] T. Stathopoulos, J. Heidemann, and D. Estrin. A Remote Code Update Mechanism for Wireless Sensor Networks. Technical report, 2004.

[87] M. Stolikj, P. J. L. Cuijpers, and J. J. Lukkien. Patching a patch — software updates using horizontal patching. In *2013 IEEE International Conference on Consumer Electronics (ICCE)*, pages 647–648, 2013.

[88] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization. 07 2003.

[89] Dingwen Tao, Sheng Di, and Franck Cappello. Exploration of pattern-matching techniques for lossy compression on cosmology simulation data sets. pages 43–54, 10 2017.

[90] Crossbow Technology. Mote In-Network Programming User Reference Version 20030315. Crossbow Technology, Inc.

[91] Walter F Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems (TOCS)*, 2(4):309–321, 1984.

[92] David Tracey and Cormac Sreenan. Oma lwm2m in a holistic architecture for the internet of things. pages 198–203, 05 2017.

[93] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

[94] Y. Tseng, S. Ni, Y. Chen, and J. Sheu. The Broadcast Storm Problem in a Mobile Ad Hoc Network. *Wireless Networks*, 8:153–167, 2002.

[95] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

[96] B. Wang, Y. Chen, H. Gu, J. Yang, and T. Zhao. Two Energy-Efficient, Timesaving Improvement Mechanisms of Network Reprogramming in Wireless Sensor Network. In *Embedded Software and Systems*, pages 473–483. Springer Berlin Heidelberg, 2005.

[97] C. Wilson. Sensors in medicine. *The Western journal of medicine*, 1999.

[98] Minoru Yoshida, Kazuyuki Matsumoto, Qingmei Xiao, Xielifuguli Keranmu, Kenji Kita, and Hiroshi Nakagawa. Extracting corpus-specific strings by using suffix arrays enhanced with longest common prefix. In *Information Retrieval Technology*, pages 360–370, Cham, 2014. Springer International Publishing.

[99] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli. Secure firmware updates for constrained iot devices using open standards: A reality check. *IEEE Access*, 2019.