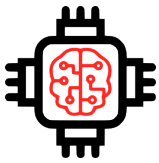
TUHH
Hamburg University of Technology



Persistent Live Firmware Patches for Cortex-M7 based Microcontrollers

Bachelor's Thesis

submitted by

Fredrik Wilke

born in Hamburg, Germany

Smart Sensors
Technische Universität Hamburg

Juli 2025

1. Reviewer : Prof˙ Dr-Ing˙ Ulf Kulau
2. Reviewer : Yannick Loeck, M.Sc.

Affidavit

I hereby declare in lieu of an oath that I have produced this Bachelor's Thesis with the title ,,Persistent Live Firmware Patches for Cortex-M7 based Microcontrollers" independently and without unauthorized outside help. I have not used any sources or aids other than those indicated, and I have marked any quotations, either verbatim or in spirit. The work has not been submitted in the same or a similar form to any examination authority.

Hamburg, July 28, 2025
Place, Date

Fredrik Wilke

Kurzfassung

Die Wartung von Software auf Kleinsatelliten bleibt eine zentrale Herausforderung, für die es bislang keine vollständig zufriedenstellende Lösung gibt. Angekommen im niedrigen Erdorbit (LEO) stehen CubeSats vor erheblichen Einschränkungen: begrenzte Uplink-Bandbreite, kurze Verbindungszeiten zu Bodenstationen und ein strenges Energiebudget. Im Rahmen dieser Arbeit wurde ein Live-Patching-Framework für einen auf dem STM32H753 basierenden CubeSat entwickelt. Das System wurde für die CubeSat-Mission der Smart Sensors Group der TUHH konzipiert und erlaubt die Erstellung, Übertragung, Anwendung und dauerhafte Speicherung von Firmware-Patches. Zum Einsatz kommen HDiffPatch und LZMA zur effizienten Komprimierung, ein eigens entwickelter Semantikanalysator zur Erkennung von geänderten Funktionen sowie der Debug Monitor des STM32H753 für die Ausführung der Patches. Das System kommt ohne Instrumentierung oder Anpassung der zu patchenden Firmware aus und lässt sich an unterschiedliche Compiler-Toolchains anpassen. Die Evaluation zeigt, dass eine durchschnittliche Kompressionsrate von 5,9% erreicht wird und der Overhead bei der Patchausführung zwischen 127 und 252 CPU-Zyklen liegt.

Abstract

Software maintenance for small satellites remains a critical yet unresolved challenge. Once deployed into Low Earth Orbit (LEO), CubeSats are constrained by limited uplink bandwidth, short ground contact durations, and strict energy budgets. This thesis addresses the development of a live patching framework for a STM32H753-based CubeSat. The proposed system enables the generation, transmission, application and persistence of firmware patches for the TUHH Smart Sensors Group's CubeSat mission. It uses HDiffPatch and LZMA for compression, a custom semantic analyser to identify semantic changes between firmware versions, and the STM32H753's Debug Monitor for patch execution. The implementation does not require code instrumentation or other modifications to the firmware it patches and is designed to be adaptable to different compiler toolchains. The evaluation of the system shows that it achieves a compression ratio of 5.9% and that the patch execution overhead is between 127 and 252 CPU cycles.

Contents

List of Figures

List of Tables

Acronyms

ADC Analog to Digital Converter

API Application Programming Interface

BWT Burrows-Wheeler Transform

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DCache Data Cache

DWT Data Watchpoint and Trace Unit

eBPF Extended Berkeley Packet Filter

ELF Executable and Linkable Format

FMC functional memory controller

FPB Flash Patch and Breakpoint Unit

FSE Finite State Entropy coding

GPIO General Purpose Input/Output

GOT Global Offset Table

HAL Hardware Abstraction Layer

ICache Instruction Cache

IETF Internet Engineering Task Force

IoT Internet of Things

IR Intermediate Representation

JTAG Joint Test Action Group

LEO Low Earth Orbit

LZMA Lempel Ziv Markov Algorithm

NVIC Nested Vectored Interrupt Controller

PIC Position Independent Code

PLT Procedure Linkage Table

PPM Prediction by partial matching

RAM Random Access Memory

RLE Run Length Encoding

RTOS Real-Time Operating System

RTL Register Transfer Language

SoC System on Chip

SRAM Static Random Access Memory

SUIT Software Updates for Internet of Things

SWD Serial Wire Debug

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

zstd ZStandard

1 Introduction

In recent years, the trend toward smaller and less costly satellite platforms has opened new opportunities for spaceborne research. Especially CubeSats, satellites based on cubic units of 10 cm x 10 cm x 10 cm (1U), have become increasingly popular for scientific missions. However, these systems impose strict limitations on available uplink and downlink bandwidth, onboard power consumption, and computational resources. Once deployed in LEO, connectivity is limited to the satellite's short passes over ground stations, typically no longer than ten minutes. These constraints make traditional software maintenance approaches, such as full-image firmware replacement, impractical.

Given the prolonged duration and high energy cost of uploading complete firmware images, software updates on such systems must be minimized in size. At the same time, safety and mission availability must be maintained.

While many systems exist to facilitate software updates for embedded systems, the unique environment of CubeSats require a specialized approach that is tailored to the mission at hand. The mission at hand is the TUHH Smart Sensors Group's effort to develop a 1.5U CubeSat for educational and research purposes. The CubeSat is based on EnduroSat's 1.5U platform and will be equipped with a STM32H753 microcontroller as well as several sensors.

1.1 Problem Statement

This thesis addresses the development of a comprehensive firmware upgrade mechanism for a STM32H753 based CubeSat. The mechanism should encompasses the full lifecycle of a patch: Given an old and a new firmware image as well as the necessary metadata, the system should be able to generate a minimal patch that can be applied in orbit. Applying the patch should not require the in orbit system to reboot. Yet, the patch should also be persistent. The new firmware image should eventually be stored in non-volatile flash memory and become the new boot target.

At the same time the upgrade mechanism should be resource efficient, meaning that it should consume minimal bandwidth, CPU cycles, Random Access Memory (RAM) and flash memory. Also, it should not impose limitations or requirements on other software components for the satellite and should be easily integrable with different toolchains. Lastly, the upgrade mechanism needs to be robust against failures, such as power loss during the upgrade process, and should not compromise the safety of the satellite.

1.2 Solution Strategy

To address these challenges, a literature review was conducted to identify existing solutions. This revealed examples based on which a system optimized for the specific requirements of this project was implemented. Different options for key components of this system were evaluated,

including the choice of compression algorithm.

The remainder of this document is structured as follows. Chapter 2 gives an overview of fundamental concepts relevant to the upgrade mechanism and related work. Chapter 3 presents the design and implementation of the upgrade mechanism. Chapter 4 evaluates the performance of the system depending on different design choices. Chapter 5 summarizes the findings and provides an outlook on future work.

2 Fundamentals

This chapter describes concepts fundamental to the design and implementation of the firmware upgrade mechanism for the STM32H753 based CubeSat. It provides an overview over the satellite mission and its hardware and software preconditions in the first section. This is followed by a review of related work on firmware upgrade techniques for embedded systems, including delta compression algorithms and compression algorithms in the second section. In the third section basic concepts relevant for semantic change detection such as compiler's intermeditate representations are introduced. Finally, an overview on handling the STM32H753's state across reboots is given in the fourth section.

2.1 Project Context and Preconditions

The Smart Sensors Group's satellite mission will use a 1.5U CubeSat platform provided by EnduroSat[**EndurosatSpec**]. The mission is intended for research and education purposes. The payload will consist of a set of sensors, which are still under definition at the time of writing.

The satellite will operate in LEO at an altitude below 2000 km. To maintain their altitude, CubeSats in LEO need to travel at speeds of approximately $7.8\,\mathrm{km\,s^{-1}}$ to counteract atmospheric drag and gravity. Thus, they typically have a low orbital period of around 90 min to 120 min. This means that the satellite will circle the Earth more than 10 times a day and that the periods in which a ground station can make contact with the satellite can be below 10 min per pass. Furthermore, the already defined radio transceiver, EnduroSat's UHF Transceiver Type II, supports uplink speeds ranging from 1200 baud/s to 19 200 baud/s. It offers a choice of modulation schemes with which a maximum data rate of $38\,400\,\mathrm{bit\,s^{-1}}$ can be achieved[End24]. Even with the most performant modulation scheme and the highest baud rate, the maximum data rate is still very low compared to terrestrial networks. This means that any communication with the satellite needs to be as efficient as possible and that firmware upgrades become a challenge.

2.1.1 Hardware Preconditions

The main controller of the satellite will be an STM32H753. This System on Chip (SoC) is centered around an Arm Cortex-M7 processor.

It comes with a total of 2MB of flash memory and 1MB of RAM.

Flash Memory

The flash memory is used to store the program and is divided into two flash banks of 1MB each. The two flash banks are by default accessible through the address ranges 0x080... (flash bank 1) and 0x081... (flash bank 2). However, the device comes with a system to switch the two flash banks so that flash bank 1 can then be referenced as 0x081.. and flash bank 2 as 0x080.... This feature allows the device to switch between different firmware images[STM09]. The switch

requires the device to perform at least a power domain D1 reset of the CPU and its closest peripherals [STM09]-(4.3.14 FLASH reset and clocks).

The device is divided into multiple power domains which can be independently controlled. Specifically, it has a D1 domain which includes the Cortex-M7 processor, the functional memory controller (FMC), the closest Static Random Access Memory (SRAM) unit, a QUADSPI external flash interface and the two flash banks. Domain D2 contains more SRAM units as well as peripherals for connectivity such as Universal Serial Bus (USB) and Ethernet interfaces. The D3 domain contains another SRAM unit, low power timers and an Analog to Digital Converter (ADC).

To write to the Flash memory there are two options. Firstly, it can be externally programmed through a debug interface. The STM32H7 supports both Joint Test Action Group (JTAG) and Serial Wire Debug (SWD) which are widely supported among programming and debugging hardware[STM09]. Secondly, the flash can be programmed from software running on the device itself. However, flash writes can only be performed on whole Flash pages. The STM32H753's flash banks are divided into 8 flash pages each. Thus, each of these pages has a size of 128 kB. Furthermore, the programming process can only switch the state of each flash memory cell from 1 to 0 or leave it at 1. It can not change a 0 to 1. To change a 0 to 1 in flash memory, the flash page first needs to be erased so that all its cells hold the value 1. Then, the program operation can be performed. Also, the STM32H7 family only allows the application running on it to write to the flash bank which is currently not being executed from. Thus, the running firmware can not modify itself directly[STM09]-(4.3.9 FLASH program operations). These are important factors in designing a patching system. They set the task at hand apart from patching in the context of non-embedded systems where applications are executed from RAM.

Debug Infrastructure

As the STM32H7 contains a Cortex-M7 processor it features its debug infrastructure. Among other debugging features the Cortex-M7 has a DebugMonitor. As opposed to tradional mechanisms that rely on halting the processor when a debug event(e.g. a breakpoint being hit) occurs, the DebugMonitor runs an interupt handler when such an event occurs. From the interrupt handler it can be controlled how to proceed after the debug event. It can issue step commands to go to the next instruction after the one that was executed when the debug event fired or let the program resume. This feature relies heavily on the priorities defined for different interrupts in the Cortex-M7's Nested Vectored Interrupt Controller (NVIC). This becomes evident when setting breakpoints in interrupt handlers. If the priority of the DebugMonitor exception is lower than the execution priority of the interrupt handler then the DebugMonitor interrupt handler is not executed and a HardFault is generated. Similarly, the DebugMonitor interrupt handler can be interrupted and preempted by interrupt handlers of interrupts with a higher priority. While this requires care when setting breakpoints, it also allows to have time critical interrupt based logic running such as network drivers while debugging lower priority logic.
Unlike earlier Cortex-M processors the M7's Flash Patch and Breakpoint Unit (FPB) does not feature a flash patch functionality that allows to redirect access from one memory location in flash to another arbitrary location.

2.1.2 Software Preconditions

To interact with the different hardware elements of the STM32H7, ST Microelectronics offers a Hardware Abstraction Layer (HAL) library and middleware packages to simplify application development. This library provides a C-Application Programming Interface (API) for many tasks that involve communication with hardware peripherals of the STM32H7. For example, it provides functions to read and write to the flash memory, to configure the NVIC and to handle General Purpose Input/Output (GPIO). In cases not covered by the HAL library, configuration and interaction with hardware peripherals such as the FPB is done through memory mapped registers. These behave like any other memory locations within the address space of the STM32H7 and are documented in the reference manual of the STM32H7 [STM09]. It also needs to be mentioned that the HAL library does not provide abstractions for the configuration of the Cortex-M7 CPU configuration. For this, special header files provided by Arm offer configuration register definitions.

To compile, link and deploy an application for the STM32H7, multiple toolchains can be used. The default toolchain that is provided for the STM32H7 by its manufacturer, ST Microelectronics, as part of its STM32Cube software ecosystem is GNU Tools for STM32. It is based on the ARM GNU toolchain and contains patches helpful for embedded systems development[Lima]. However, other toolchains for Arm Cortex-M processors can also be used.

The instruction set used by the STM32H753 is the Armv7-M Thumb instruction set. It contains both 32-bit and 16-bit instructions. Especially for the 16-bit instructions it is common to not encode literals directly in the instruction but to use a register to hold the literal value or to load it from a memory location using a register relative instruction. For this, literals are stored as 32-bit words in the flash memory after a function's last instruction. Groups of such literals are refered to as *literal pools*.

These software and hardware preconditions are an important pillar for this project to implement a firmware patching system for the STM32H7. The next section will provide an overview of existing techniques and approaches to firmware upgrades for embedded systems which are also fundamental for the project.

2.2 Firmware Upgrade Techniques

Research on firmware upgrades for embedded devices often has to take further constraints into account. An overview by Brown and Sreenan highlights that for Internet of Things (IoT) systems remote firmware upgrade techniques need to address four issues: dissemination of the upgrade, traffic reduction during dissemination, upgrade execution environment and fault detection and recovery [BS13].

2.2.1 Upgrade dissemination

As Brown and Sreenan note, on wireless sensor networks and embedded systems, a major challenge is efficiently disseminating software updates to many devices over unreliable or constrained links. Numerous protocols have been developed to address this issue.

Furthermore, security considerations are paramount. As in any data transmission process, the integrity of the data being transmitted is crucial. Especially in the context of firmware updates, it is important to ensure that the data being transmitted is not corrupted as a corrupted firmware image can lead to a non-functional device. Also, authenticity is important to ensure that the firmware image received by the device was intentionally sent by the owners of the device and not by an attacker.

To address these concerns in a standardized way, the Internet Engineering Task Force (IETF) developed the Software Updates for Internet of Things (SUIT) architecture, formalized in RFC 9019 [Mor+21]. SUIT defines a secure boot and update mechanism specifically tailored for constrained devices, where traditional cryptographic protections are often too costly in terms of memory or compute resources. Instead of relying solely on external transport-layer security, SUIT introduces a manifest format that accompanies each firmware update. This manifest describes the structure of the firmware image and includes metadata such as version numbers, digests (hashes) for integrity checking, and cryptographic signatures to authenticate the update source. By verifying the manifest before executing any update, the device can ensure both integrity and authenticity of the firmware, regardless of the transmission path. This approach enables secure update workflows even in lossy or adversarial network environments and supports rollback prevention and multi-image update coordination.

However, upgrade dissemination is not a focus of this work. In the context of this thesis, only a single device — a CubeSat — is upgraded at a time, and the communication protocol used for transmission, the standardized CubeSat Space Protocol, is already predefined and supports authentication as well as integrity checks using Cyclic Redundancy Check (CRC)[lib]. Therefore, the techniques for multi-node dissemination are not applicable here and security should be considered when implementing the communication logic for the satellite.

2.2.2 Traffic reduction during dissemination

The most obvious way to perform a software upgrade is to transfer the whole upgraded version of the software to the device which is supposed to be upgraded. However, the upgraded version might only differ from the original version minimally. In this case, an artefact that describes the difference (diff or patch) can be transfered to the device which then constructs the updated version of the software from the original version and the diff. As the communication link to the satellite has very limited bandwidth, it is important to minimize the amount of data that needs to be transferred. In the case of the STM32H753 a full firmware image can be up to 1MB in size, while patches are usually limited to very few code changes which can be represented in a few hundred bytes. This can be achieved by using delta upgrades, which only transfer the differences between the old and the new firmware image. However, this concept is not only applied in the context of embedded systems. Linux operating systems include the commands `diff` and `patch` which can be used to create and apply patches to files [GNU24].

Oro gives an overview of the use of these techniques for satellite systems [Oro18]. He mentions missions such as MINISAT01 that were able to perform both full image upgrades and differential upgrades. He highlights that the latter is particularly useful for missions with limited bandwidth such as the one at hand. Differential upgrades were reported to reduce the time required for an upgrade from days to hours.

Depending on the software architecture used for the device, there are more ways to reduce the size of the upgrade package. Coelho et al. propose a modular software architecture with their NanoSat MO Framework that allows "Apps" for satellites to be deployed and updated

independently [CKM17]. These apps rely on two services running on the satellite: a "Package Management" service that manages the apps present on the device and an "Apps Launcher" service that manages the execution of the apps. In case of an update, only the app that is being updated needs to be transferred to the device. Zandberg et al. propose a similar architecture for IoT devices with their FemtoContainers [Zan+22].

Delta Compression Algorithms

Delta upgrades are a well explored field of research. They encode the difference between two versions of a document in an artefact which later can be used to produce the newer version of the document based on the older.

xdelta3 is an example of a widely used delta compression tool that implements a version of the VCDIFF algorithm (RFC 3284) for encoding the difference between two binary files [Mac]. The resulting patch artefact is a compact, binary file composed of a series of "copy", "add", and "run" instructions that describe how to construct the target file from the source. "Copy" operations reference byte ranges from the original file that should be copied to the new file, while "add" inserts literal byte sequences, and "run" denotes repeated bytes. Figure 2.1 shows an example of a RFC 3284 compliant patch artefact. These instructions form a patch script interpreted by the xdelta3 decoder to reconstruct the new file.

There are several other algorithms that can be used to create delta upgrades, such as bsdiff [**bsdiff**] and HDiffPatch [Sona]. Comparisons of these algorithms highlight that HDiffPatch's reconstruction mechanism is particularly resource efficient [Sun+24].

Figure 2.1 COPY is encoded as a pair: length and offset from the source window. Target window copies are allowed, using indices starting from `len(source)`.
ADD encodes a literal sequence, and
RUN encodes repetition of a byte.

```
1: a b c d e f g h i j k l m n o p                                  // Source string
2: a b c d w x y z e f g h e f g h e f g h e f g h z z z z          // Target string
3: COPY 4, 0                                          // Copy 4 bytes from offset 0
4: ADD 4, w x y z                                        // Add 4 bytes: w x y z
5: COPY 4, 4                                          // Copy 4 bytes from offset 4
6: COPY 12, 24                          // Copy 12 bytes from offset 24 (in target)
7: RUN 4, z                                                    // Add 4 bytes of z
```

HDiffPatch's algorithm is based on the same priciple as xdelta3, with some additional features and a different encoding scheme which is only partially compatible with RFC 3284. The additional features include that it allows merging of multiple copy instructions into one even if they are not adjacent in the source file. This works by adding a diff block besides the control block of the patch artefact. The control block contains the instructions while the diff block contains further modifications that need to be done to the target file after the instructions have been applied. These modifications are encoded in Run Length Encoding (RLE). An example of an RLE encoded word is "aaaabbbc" which is encoded as "4a3bc". This way there is not a big overhead caused in the diff block by a long copy instruction of n bytes because the diff block contains only n0. At the same time the number of copy instructions can be reduced [Sun+24].

Compression Algorithms

To further reduce the size of the upgrade package compression is widely used. When choosing a compression algorithm, there are many factors that need to be weighed against each other such as compression ratio, decompression speed, compression speed, memory usage during decompression, lossy vs lossless compression etc. The definition of the compression ratio is given in as defined by Salomon and Motta [SM09]:

$$\text{Compression ratio} = \frac{\text{size of the output stream}}{\text{size of the input stream}} \tag{2.1}$$

Especially as the STM32H7 is a resource constrained device with significant requirements to its availability, the ressources consumed by decompression are an important factor. Not only the decompression time but also the RAM usage during decompression are important in choosing a compression algorithm. Gupta et al. provide an overview of compression algorithms considered as modern in 2017 [GBK17]. Specifically, they compare LZMA, Bzip2, Deflate and PPM. Lu et al. considers the same algorithms except for PPM and adds zstd[Lu+20]. Salomon and Motta explain the algorithms except zstd in detail in their book on data compression[SM09]:
LZMA is a lossless dictionary based compression algorithm which has its root in the LZ77 compression algorithm. While reading the input stream a dictionary based compression algorithm builds a dictionary of previously seen characters or strings and replaces them with references to the dictionary. After the dictionary stage, it includes an entropy coding scheme that leverages the frequencies of characters and words in the input stream to further compress the data.

Bzip2 is a block compression algorithm that uses the Burrows-Wheeler Transform (BWT) on blocks of the input stream. The BWT is a lossless compression algorithm that chooses a permutation of the input stream that results in a better compression ratio when applying RLE, move-to-front coding and Huffman coding afterwards. Move-to-front coding is a technique that encodes a string by replacing each character with its index in a list of characters that is updated after each character is processed. The updated list has the most recently used character at the front and the least recently used character at the back. This way, local differences in the frequencies of characters can be exploited to increase the compression ratio. Huffman coding is a statistical compression algorithm that builds a binary tree in which the leafs correspond to the characters from the input stream. The tree is built by always merging the two least frequent characters into a new node until only one node is left. The code for each character is then the path from the root to the leaf of the character. This results in longer codes for less frequent characters and shorter codes for more frequent characters.

Deflate is a combination of the LZ77 algorithm and Huffman coding. It uses the LZ77 algorithm to build a dictionary of previously seen strings and replaces them with references to the dictionary. Then, it applies Huffman coding to the resulting stream to further compress it.

PPM is a lossless statistical compression algorithm that builds a model of the input stream based on the frequencies of characters and their preceding context. If the default context of length n is not present in the model, it uses the context of length n-1 and so on until it finds a context that is present in the model. Then, it replaces the character with the code for the character in the model.

zstd is also lossless and combines the LZ77 dictionary based compression with Huffman coding and Finite State Entropy coding (FSE). FSE can be seen as a more efficient variant of arithmetic coding which uses floating point numbers to represent strings based on their frequencies of occurence. Instead of floating point numbers, FSE uses codes of variable length together with a state machine to represent characters [Fac25a].

Gupta et al.'s comparison of the compression algorithms shows that on the full Silesia corpus of documents PPM achieves the best compression ratio of 14.11% (7.088 by their definition of compression ratio) for the PPMonstr implementation of PPM [GBK17]. However, not all of the data included in the Silesia corpus is relevant for the project at hand. Looking only at the results for binary executables, the PPMonstr implementation of PPM still achieves the highest compression ratios of 23.35% and 32.44% for the Mozilla Firefox and OpenOffice executables respectively. The PPmd implementation of PPM achieves lower compression ratios of 30.62% and 40.58% for the same files. Notably, the compression ratios achieved by LZMA are 26.08% and 39.44% for the same files, so between the two PPM implementations. The decopression speeds reported show that Deflate is the fastest on an Intel(R) i5-3230M processor with an average of 91.3 MB/s. LZMA achieves 57.7 MB/s, while PPMd and PPMonstr achieve 5.7 MB/s and 0.8 MB/s respectively. Bzip2 achieves 17.7 MB/s. These results draw a clear picture of the advantages and disadvantages of the compression algorithms and suggest that LZMA is the most suitable candidate for embedded systems.
Lu et al.'s comparison highlights LZMA as the algorithm achieving the best compression ratio with zstd being behind LZMA and Bzip2. However, the decompression speed is not part of the results and the data used for the comparison were images and text files. Further research on the performance of zstd is limited. A comparison by Rovnyagin et al. shows similar results to Lu et al.' for compression ratios and compression speeds but also does not provide decompression speeds [Rov+23]. This is a significant gap in the research as decompression speed is crucial for determining the suitability of a compression algorithm for embedded systems.
Furthermore, zstd's documentation mentions that it includes a training mode to improve its compression ratio for files in the range of 1KB[Fac25b]. This is a feature that is not present in the other algorithms and might be useful in the context of firmware patching as the patches are usually small in size.

Similarity Improvement

Even with compression, the size of the patch strongly depends on the similarity of the old and the new firmware images. Because of that there is research on how to increase the similarity while maintaining the desired functionality of the patch. Arakadakis et al. propose four factors that affect the size of the binary patch (or patch script) [Ara+21]:

- Function shifts: When a functions length changes the start addresses of the subsequent functions changes. This results in absolute or relative references to these functions to change as well which causes the patch to grow in size.
- Global variables shifts: Global variables reside in the .data or .bss section of the program. When global variables are added or removed the address of the subsequent variables changes. Thus, all references to these variables change which results in more ADD instructions in the patch.
- Relative jumps (inside functions): Depending on the memory model used, functions may

call each other not by their full addresses but by their relative offset from the calling instruction. These relative calls are also changed when code has been added between them and their target.

- Indirect addressing: Certain architectures including ARM support indirect branches in which the branch target is either an address stored in a register or at another memory location. In the case of an address in a register, the computation steps of this address might change when the address of a function changes. An example is when a 32 bit address is written into a register by the means of two writes of 16 bit literals and a shift in between. In the case of a 32 bit address being directly loaded from a memory word W into a register or an indirect branch to an address stored in W the memory word W needs to be patched.

Arakadakis et al. introduces multiple techniques to ameliorate these factors. Notably, they propose regions of unused memory between functions (Slop regions) and multiple techniques for effectively storing jump target addresses and for modularizing the firmware image. Kachman et al. also highlight these methods[KBM19].

Both groups also stress that Position Independent Code (PIC) solves many of the issues that arise from function shifts and global variable shifts. However, both mention that it also causes a performance overhead. This is because PIC relies on Global Offset Tables (GOTs) or Procedure Linkage Tables (PLTs) to resolve the addresses of global variables and functions. Every function call or access to a global variable has to first look for the address of the function or variable in the GOT or PLT and then call it. This is in contrast to direct calls which can even be implemented by a single 24-bit relative jumps in case of the STM32H7 because its image is constrained by 1MB flash banks.

Kachman et al. also refers to a work by Li et al. that presents a compiler modified for minimizing the difference between two versions of a program [Li+07]. This is done by modifying the algorithm that allocates registers to variables. By default a change in the lifetime of a variable might cause the register assigned to it to change. To maintain as much similarity as possible, Li et al.'s system inserts MOV-instructions between semantically changed and unchanged areas of the program so that in the semantically unchanged areas the assignment of registers to variables remains the same.

Compiler Optimizations & Binary Semantics

In the scenario at hand compiler optimization plays a crucial role as a tool to make use of the limited resources available on the STM32H7 as efficiently as possible. Specifically, the optimizations that GCC combines under the -Os flag are important as they optimize the program for size [**gnuOpt**].

However, compiler optimization might have a detrimental effect on the size of the patch artefact as Tables 2.1 and 2.2 illustrate for the example function `compute_checksum` from Listing 2.1. With the -O0 flag set, the compiler generates a function that is a direct equivalent of the source code. This can be seen in Table 2.1. The added line of source code results in seven additional lines of assembly code. Meanwhile, with the -Os flag set, the compiler optimizes the function so that the patched and unpatched versions only have one line of assembly code in common. This is shown in Table 2.2.

```c
int compute_checksum(const uint8_t* data, int len) {
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += data[i];
    }
    return sum;
}

int compute_checksum_patched(const uint8_t* data, int len) {
    if (len < 0) len = 0; // Patch: avoid negative lengths
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += data[i];
    }
    return sum;
}
```

Listing 2.1: Patched and unpatched versions of example funtion `compute_checksum`

compute_checksum	compute_checksum_patched
b480 push {r7}	b480 push {r7}
b085 sub sp, #20	b085 sub sp, #20
af00 add r7, sp, #0	af00 add r7, sp, #0
6078 str r0, [r7, #4]	6078 str r0, [r7, #4]
6039 str r1, [r7, #0]	6039 str r1, [r7, #0]
	683b ldr r3, [r7, #0]
	2b00 cmp r3, #0
	da02 bge.n pc + 4
	2300 movs r3, #0
	603b str r3, [r7, #0]
	e000 b.n pc + 0
	bf00 nop
2300 movs r3, #0	2300 movs r3, #0
60fb str r3, [r7, #12]	60fb str r3, [r7, #12]
2300 movs r3, #0	2300 movs r3, #0
60bb str r3, [r7, #8]	60bb str r3, [r7, #8]
e00a b.n pc + 10	e00a b.n pc + 10
68bb ldr r3, [r7, #8]	68bb ldr r3, [r7, #8]
687a ldr r2, [r7, #4]	687a ldr r2, [r7, #4]
4413 add r3, r2	4413 add r3, r2
...	...

Table 2.1: Assembly implementation of `compute_checksum` before (left) and after (right) the patch with compiler optimizations disabled (-O0). Green entries are newly introduced, red entries were removed.

compute_checksum -Os	compute_checksum_patched -Os
2300 movs r3, #0	4603 mov r3, r0
4602 mov r2, r0	2900 cmp r1, #0
4618 mov r0, r3	bfac ite ge
b510 push {r4, lr}	1842 addge r2, r0, r1
428b cmp r3, r1	1c02 addlt r2, r0, #0
db00 blt.n pc + 0	2000 movs r0, #0
bd10 pop {r4, pc}	4293 cmp r3, r2
	d100 bne.n pc + 0
5cd4 ldrb r4, [r2, r3]	4770 bx lr
3301 adds r3, #1	f813 1b01 ldrb.w r1, [r3], #1
4420 add r0, r4	4408 add r0, r1
e7f8 b.n pc - 16	e7f8 b.n pc - 16

Table 2.2: Assembly diff of `compute_checksum` before and after the patch. Branch instructions are shown in PC-relative form. Green entries were added in the patch. Red entries were removed.

2.2.3 Upgrade Execution Environment

Once a patch has been deployed to the device, there are multiple steps that need to be performed for the patch to be executed. These include changing the control flow of the program to use the new instructions and in many cases also providing a runtime environment in which the patch can safely run.

Live Control Flow Modification Techniques

After updated instructions have been deployed to the device, the control flow of the program must be changed to use the new instructions. This is a non-trivial task, as it requires redirecting the control flow of *running* firmware. There are several approaches to this problem. Zhou et al. [Zho+24] propose a method that relies on hardware breakpoints to change the control flow of the program. These breakpoints are placed by the program itself at the location where control flow is to be diverted to the patched code. When the program reaches the location of the breakpoint, an interrupt is generated and the jump to the patch code is performed in the interrupt handler.

A similar approach is native to the Cortex-M4 family of processors. They include a FPB which allows the program to define positions in ROM for which fetch operations will be redirected to another memory address[Lim09]. It has been shown that on top of this feature, a live patching mechanism can be implemented. Niesler et al. propose HERA, a system that redirects control flow by redirecting fetch operations to special jump instructions refered to as "trampolines" that redirect the control flow to the patch code [NSD21]. Any approach that relies on hardware breakpoints is limited to the number of hardware breakpoints that are available on the device. The device at hand, the STM32H7, has an FPB that can be configured to have up to 8 hardware breakpoints. An alternative approach is to implement a software-based control flow change mechanism. Salehi and Pattabiraman [SP24] propose a method that places trampoline functions in strategic positions of the original code which can be utilized at runtime to jump to code provided by a patch. This approach is also refered to as code instrumentation and is not limited by the number of hardware breakpoints available on the device. However, it is limited by the number of trampoline functions included in the original code.

Runtime Environments for Patch Execution

While embedded systems generally support executing binary instructions not only from ROM but also from RAM, this is not the only approach used for running patch code. There are multiple other approches with different trade-offs. He et al. [He+22] include a runtime for Extended Berkeley Packet Filter (eBPF) instructions which is used to execute the instructions of the patch in a sandboxed environment. This has the advantage that patches are universally compatible with the runtime, as they are not tied to a specific architecture, which not only decreases the development time but also simplifies the verification of the patch. Zandberg et al. [Zan+22] share the objective of general applicability to different architectures, but furthermore embeds the patch runtime into a Real-Time Operating System (RTOS) to allow for a more seamless integration into the system. This allows their system to make use of established secure update mechanisms such as SUIT.

2.3 Semantic Change Analysis & Detection

When a differential upgrade is deployed and an immediate swap of flash banks is supposed to be avoided because of the overhead it causes, the semantic differences between the original and updated version of the software need to be determined. An example for why this is necessary is that if function g is located after function f in ROM and in the updated version function f is longer than in the original version, the address of function g changes. With this, all the references to g change. That way it might appear that there are many changes between the versions while there is actually only one and the original software can continue to run with only the new version of function f being executed whenever the old version of f was supposed to be executed.

To detect which parts of the software changed semantically there are many techniques and tools. Apart from these tools which are primarily designed for reverse engineering of programs with unknown source code, there are tools that rely on changes in the source code to identify semantic changes such as GumTree[Gum25]. However, such tools require special treatment of changes in preprocessor macros and also do not account for effects of compiler optimization. An example is function inlining. If function f is inlined by the compiler in five locations then all these locations need to be patched. If a tool that compares only the source code is used, then additional information from the compiler needs to be taken into account to identify the locations in which changes of f need to be applied. Another technique is to use artefacts generated by the compiler to determine the semantic difference between two software versions. This approach will be discussed in the following subsection.

2.3.1 Compiler Intermediate Representations

The two compilers officially supported by ARM Cortex CPUs are GCC and LLVM based. Both of the compilers produce Intermediate Representation (IR) of the source code before it is converted into binary instructions and can also emit these for external use. Furthermore, compilers iterate over these IRs multiple times to optimize the program and eventually generate the final binary instructions. Each of these iterations over the IR is called a pass. It is important to note that an IR file is generated for each translation unit of the source code. A translation unit is a source file

after preprocessing, meaning that all preprocessor macros have been expanded and all included files have been merged into the source file.

Cooper and Torczon classify IRs by structural organization, level of abstraction, and naming discipline. To get the required information from an IR it is especially important to consider the options in the first two categories[**CooperCompiler**]. The naming discipline is less important in this context, as it does not affect the representation of the control flow.

Structural Organization

Structural organization refers to how the IR models the control flow of a program. Cooper and Torczon introduce three categories: Graphical IRs, Linear IRs, and Hybrid IRs. Graphical IRs represent the program as a directed graph, where nodes represent instructions and edges represent control flow. Linear IRs represent the program as a sequence of instructions. Hybrid IRs combine both graphical and linear representations. Commonly, they are structured using blocks. Within blocks, the instructions are represented in a linear fashion, while the blocks themselves are connected in a graphical manner. This way, complex control flows between blocks can be expressed.
GCC's IRs GIMPLE and RTL are examples of graphical and hybrid IRs, respectively.

Level of Abstraction

Independently of the structural organization, an IR can use different levels of abstraction. It can be higher-level, meaning closer to the source code, or lower-level, meaning closer to machine instructions. It might even produce multiple IR-instruction which later can be combine into a signle machine instruction. GCC's GIMPLE is a higher-level IR which in its syntax resembles source code. RTL on the other hand is a lower-level IR. It's instructions can often be directly mapped to machine instructions. In later passes of the compiler, RTL also encodes the exact registers involved in each instruction, which is crucial for understanding differences in register usage between two versions of a program.

2.4 Device State Handling & Crash Consistency

Embedded systems are often deployed to volatile environments which might cause them to crash or lose power at any time. Thus, any firmware upgrade mechanism needs to be able to handle such situations gracefully. This is especially relevant in systems where only one flash bank is available for storing the firmware image. In this case, the firmware upgrade mechanism needs to ensure that the device can still boot after a crash during the upgrade process. In case of the STM32H7 any firmware upgrade process has the advantage that the existing firmware image does not have to be altered by the upgrade and will be used at the next boot unless the bank swap procedure is initiated.

In the context of satellite systems there is an additional parameter that needs to be considered: the power consumption of flash erase and write operations. The STM32H7's datasheet states that flash write and erase operations typically result in a current of 35 mA[STM23]. This compares to a typical operating current of 33 mA to 78 mA of the STM32H7 at 200 MHz and 3.3 V. For 400 MHz this can rise to 165 mA depending on the peripherals enabled[STM23]. This suggests an average power consumption of 0.2 W in idle mode and an extra 0.1 W for flash write

and erase operations. Given that EnduroSat's 1.5U CubeSat which will be the platform for the project at hand is stated to have 0.9 W available on average for the payload, it needs to be ensured that flash write and erase operations are timed well[End21].

3 Implementation

As previously introduced, any firmware upgrade mechanism for embedded systems can be divided into four main components: *dissemination of the upgrade*, *traffic reduction during dissemination*, *upgrade execution environment*, and *fault detection and recovery*.

Topics typically associated with *dissemination of the upgrade*, such as transmission reliability and message integrity, are not within the direct scope of this project. The only requirement placed on the dissemination mechanism is that it ensures authenticity and integrity of the upgrade artifact. These concerns will be addressed in a later stage when integrating the CubeSat Space Protocol and the upgrade mechanism into the full software stack of the satellite.

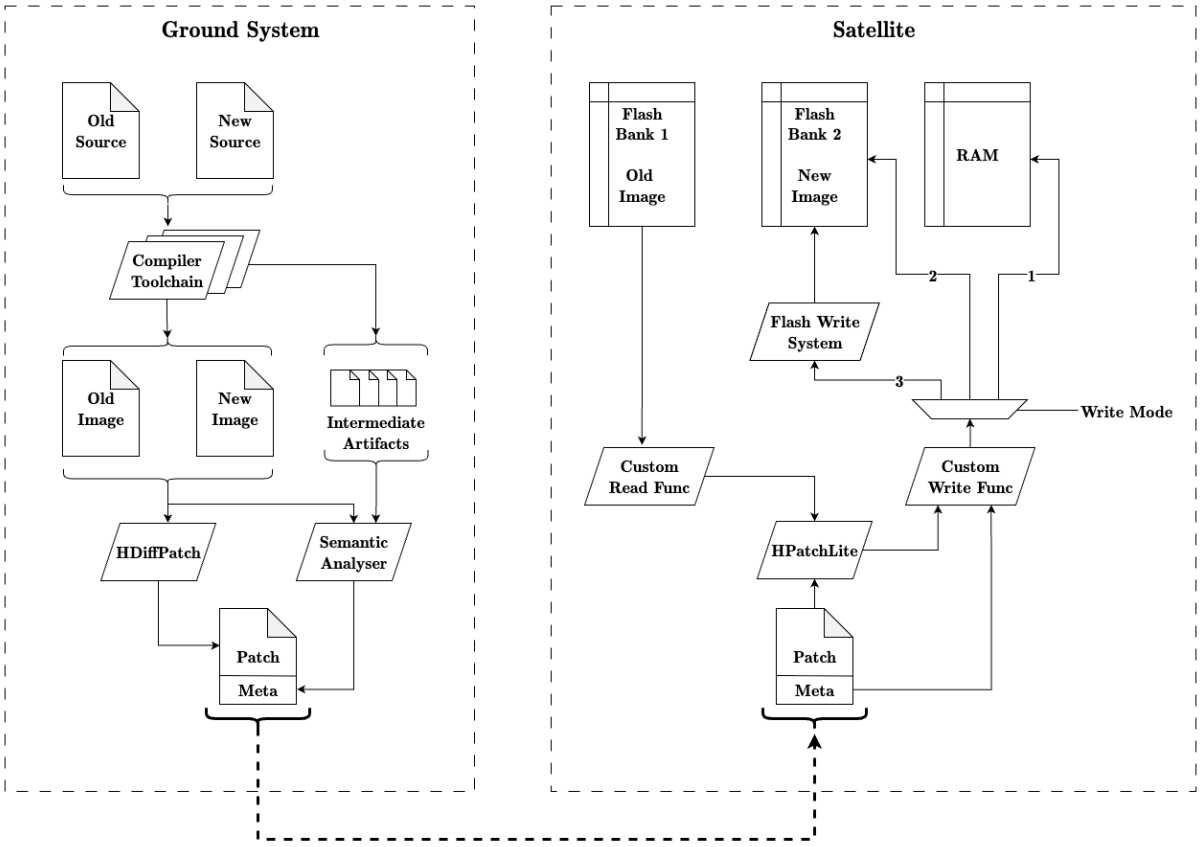


Figure 3.1: Architecture of the STM32H7 upgrade deployment mechanism. The left side shows the ground system. The right side shows the satellite system. The labels 1, 2, and 3 indicate the three methods to handle the new image on the satellite. It can be written to flash in two ways or its semantically changed functions can be written to RAM. These methods are not mutually exclusive.

Figure 3.1 provides a high-level overview of the architecture of the STM32H7 firmware upgrade deployment mechanism. The system is divided into components on the ground (left side) and

components onboard the satellite (right side).

The ground system is responsible for producing the upgrade artifact. This artifact consists of a delta patch and metadata. The delta patch is generated by `hdiffi`, the CLI tool from HDiffPatch, based on the old and the new firmware image. `hdiffi` also compresses the patch. The metadata is generated by analyzing GCC's RTL representation, the assembly code produced for each translation unit as well as map files of the images.

Once the STM32H7 receives the upgrade artifact the patch is extracted from the artifact. HPatchLite decompresseses the patch and uses it to reconstruct the new firmware image from the old firmware image. For the upgrade to become effective as soon as possible, the new semantically changed functions are written to RAM. Furthermore, the metadata is used to modify the patched functions to correctly reference functions and variables that didn't change between the old and the new firmware image. Arc 1 in Figure 3.1 shows this.
At this stage the FPB is also configured to raise a debug event when one of the patched functions is entered so that the control flow can be redirected to the new version of the function in RAM. When resources allow, the new firmware image is written to flash. This can be done in two ways:

- The entire new firmware image is written to the second flash bank in one atomic operation. This is shown as arc 2 in Figure 3.1.
- The new firmware image is written to flash bank 2 using a non-blocking flash write system which can be adapted to the available resources. This is shown as arc 3 in Figure 3.1.

The following sections will describe the implementation of the upgrade mechanisms in more details. *Traffic Reduction During Dissemination* will provide more details on the generation of the upgrade artifact. *Upgrade Execution Environment* will explain how the FPB and the Debug Monitor are used to redirect the control flow. *Device State Handling & Crash Consistency* will further explain the non-blocking flash write system and measures to handle unexpected events in the upgrade process.

3.1 Traffic Reduction during Dissemination

This section will explain the considerations and implementation details aiming to reduce the size of the upgrade artifact. First, details about the delta compression and compression algorithms used will be provided. Then, considerations regarding similarity improvement and compiler optimizations will be presented.

3.1.1 Delta Compression Algorithms

As previously noted, HDiffPatch is currently considered to be the most advanced delta compression algorithm. For this project, its version specific for constrained devices, HPatchLite, was used[Sonb]. HPatchLite is compatible with artifacts generated by HDiffPatch. Therefore, the host device which creates the patch and sends it to the satellite uses HDiffPatch's hdiffi CLI to generate a binary diff and to compress this diff. On the satellite, HPatchLite's library is used to both decompress the package and to reconstruct the new firmware image from the diff and the existing version. For this, HPatchLite provides an API that allows the programmer to specify custom functions to read the diff and the old image as well as write the new image.

3.1.2 Compression Algorithms

The compression algorithm selected for this project is LZMA. HPatchLite offers a built-in LZMA decompressor which is used to decompress the patch artifact while reconstructing the new firmware image. It does not first reconstruct the full uncompressed patch in RAM to then apply it to the old firmware image. This reduces the memory overhead of the decompression process as the decompressed diff can be sizeable for HDiffPatch (see section 4.1). HPatchLite allocates 48 kB of memory to buffer the decompressed diff as well as the new firmware image. Furthermore, the LZMA decompressor computes the memory requirements for decompressing the patch and allocates the required memory beforehand. According to the documentation, these memory requirements range between 8 kB and 32 kB plus the size of the dictionary used by the LZMA algorithm[Sonb].

3.1.3 Similarity Improvement

The size of a patch can be reduced by measures to improve the similarity between the old and the new firmware image. However, the techniques proposed to do this are not in line with the requirements of this project. Specifically, Slop regions are not considered as they bloat the image size which is not acceptable. Furthermore, PIC is not considered for compatibility reasons. This is because only the LLVM based Arm Toolchain for Embedded supports PIC while the GCC based ARM GNU Toolchain and with that GNU Tools for STM32 don't. Thus, opting for PIC would force the satellite mission to use Arm Toolchain for Embedded. Furthermore, at the time of writing Arm Toolchain for Embedded has not been released in a stable version. The available Open Beta version explicitly mentions "Position Independence" as an architectural feature which is not yet recommended to be used with the toolchain[Limb].

3.1.4 Compiler Optimizations & Binary Semantics

Compiler optimization is likely to play an important role for the mission which will use the developed upgrade mechanism. Especially the `-Os` optimization level is expected to be used to reduce the size of the firmware image as it can significantly reduce the size of the binary image compared to the default optimization level. During the course of this project, firmware images compiled with the `-Os` optimization have consistently shown a reduction in size of up to 30% and a significant reduction in patch size for the combination of HDiffPatch and LZMA compared to the default optimization level. Thus, the `-Os` optimization level is used for this project.

3.2 Upgrade Execution Environment

This section will explain the implementation details of the mechanism to redirect the control flow from the old firmware version to the changed functions of the new firmware version. Furthermore, it will explain the considerations regarding dynamic code runtimes such as eBPF.

3.2.1 Live Control Flow Changes

It has been part of the requirements to the firmware upgrade mechanism that the firmware should not have to be instrumented. To avoid this, the upgrade execution environment imitates that of the ARM Cortex-M4 family of processors which features a FPB that allows the program

to define positions in ROM for which fetch operations will be redirected to another memory address[Lim09]. This is done by using the Debug Monitor functionality of the STM32H7 which allows the program to handle interrupts generated by the device's debug infrastructure. Specifically, the upgrade mechanism defines hardware breakpoints which can be placed anywhere in address space of the device using its FPB. When the program reaches the location of a breakpoint, the debug monitor exception handler diverts the control flow to the newer version of the code.
For this, it needs to be ensured that the execution priority of the code is lower than the priority of the debug monitor exception. Otherwise, the device raises a HardFault exception when it encounters the breakpoint.

In the implementation presented here, the newer version of the code is a complete function which also includes a return instruction. In this case, the control flow does not need to be modified again after the function has been executed.
A possible alternative to this implementation is to only load the instructions that have changed semantically into RAM. In such a setting, the control flow needs to be diverted to a different location after the newer version of the code has been executed. For this, a special trampoline snippet is appended to the end of the new code. It contains a jump instruction to the location where the control flow should continue after the new code has been executed.
This option was not implemented as exploratory tests showed that many apparently small changes in the source code lead to large changes in the optimized assembly code. A notable example of this is when a new variable which is stored on the stack is added to a function. This leads to changes in both the prologue and the epilogue of the function to correctly set up the stack frame and to restore it after the function has been executed. In such a case, only the instructions before the stack pointer is modified in the prologue and after it is modified in the epilogue could be reused from the old firmware.
In addition to that, literal pools at the end of the function also need to be copied to RAM to ensure the correct execution of the new code. However, the instructions expect the literal pool to be at the very end of the function. This means that memory needs to be allocated for the patched function to also cover the literal pool at the same offset from the instructions as in the new firmware image. Another option would be to move the literal pool further forward so that it is directly after the trampoline snippet that brings the control flow back to the old firmware. However, this would require changes to every program counter relative load, store or jump instruction in the patched function which is not feasible in the context of this project.
A compiler based solution for this problem would be to replace program counter relative addressing with addressing that is relative to another register. This register could be set at the beginning of every function to point to the next literal pool. When the function is patched, the register could be set to point to the patched literal pool which could be placed arbitrarily. However, to the best of the author's knowledge, no such option exists for the ARM GNU toolchain used in this project.

3.2.2 Dynamic Code Runtimes

As the requirements for the upgrade execution environment are very specific to the STM32H7, elaborate systems to maximize the compatibility of the upgrade execution environment with different platforms are not considered. Especially virtualization using eBPF or similar techniques were not considered as they add an overhead on the complexity of the system. They would require generating eBPF code for the semantic changes between the old and the new firmware

```
1  const uint8_t rodata_array1[] = {'A', 'B', 'C', '\n', '\0' };
2  const uint8_t rodata_array2[] = {'A', 'B', 'C', 'D', '\n', '\0' };
3
4  void print_something() {
5      ...
6    printf(rodata_array1);
7    printf("This is unpatched\n");
8      ...
9  }
10
11 void print_something_patched() {
12     ...
13   printf(rodata_array2);
14   printf("This is patched and a little bit longer!!!\n");
15     ...
16 }
```

Listing 3.1: Patched and unpatched versions of example funtion `print_something`

image. This is superfluous given the fact that the STM32H7 can also execute instructions from RAM.

3.3 Semantic Change Analysis

To determine which changes between the old and the new firmware image are semantic changes, RTL, one of the intermediate representations provided by GCC is used. Specifically, the artifact produced by the final pass of the compilation process is used to determine the semantic changes. After this pass, the compiler has already performed all optimizations and the code is a direct equivalent to the machine code which will be generated afterwards.

The artifact used for semantic change analysis is not directly usable as it contains many IDs and addresses that do not encode any semantic information. It also includes the file names and line numbers of the source code. These are removed in the first stage of the semantic change analysis.

For the simplest functions the cleaning process is sufficient to prepare the artifact for a semantic comparison. However, there are several cases where additional information needs to be included into the instructions to fully capture their semantics. Specifically, load, store and jump instructions do not contain complete information about the position in memory involved in the operation.

RTL Representation
Original rodata array ref: `(insn ...[(symbol_ref/f:SI ("*.LC72") [flags 0x2] <var_decl 0x10b592ea0 *.LC72>)]... )`
Original string literal ref: `(insn ...[(symbol_ref/f:SI ("*.LC73") [flags 0x2] <var_decl 0x10ba7a090 *.LC73>)]... )`
Cleaned rodata array ref: `(insn ...[(symbol_ref/f:SI CodeLabel([CodeLabelLine(ascii, "ABCD\000")]) [flags 0x2] <var_decl >)]... )`
Cleaned string literal ref: `(insn ...[(symbol_ref/f:SI CodeLabel([CodeLabelLine(ascii, "This [...] longer!!!\000")]) [flags 0x2] <var_decl >)]... )`

Table 3.1: Original vs. cleaned RTL representation of references to global constants and string literals in a `print_something`'s literal pool. The cleaned version includes the string literal or constant value in the instruction that references it. Red indicates elements that were removed. Green indicates elements that were added to encode semantics. The uncleaned RTL representation would look the same for both the unpatched and the patched versions from Listing 3.1. The cleaned version encodes the difference between the two versions.

In the critical cases, the target involved is stored in one of the literal pools of the function. This is in turn referenced by the load, store or jump instruction by a program counter relative offset.
In case of a function call, the name of the function is included in the RTL representation of the word in the literal pool. This name can simply be included in the line that references the value in the pool to encode the semantics of the instruction. The same applies for global variables. Similarly, if the literal pool contains a constant such as the address of a peripheral register, this constant can be included into the instruction that references it.

A more complex case is that of global constants and string literals which are not stored in the literal pool but in the rodata section of the binary. Here the RTL involved with the constant might not change between the old and the new firmware image if the value of the constant changes. Still, all functions accessing the constant need to be patched when the value of the constant changes. This is because neither the value of the constant can be changed at runtime nor can the literal pool pointer to the constant be changed at runtime because both are in ROM. Furthermore, global constants and string literals can only be identified by looking at the binary image, the Executable and Linkable Format (ELF) file or the assembly code produced for each translation unit. This is because the compiler uses special labels for them which are not included in the map files and are only unique within the translation unit (e.g. "*.LC72"). However, the assembly file uses the same special labels as the RTL representation which makes the identification process easier than finding the literal pools accessing it in the ELF file or in the binary image as both are binary formats. The definition of the string literal or global constant is then included in the RTL representation of the instruction that accesses it. Table 3.1 shows how constant values are included into literal pool entries to make them comparable. The literal pool entries are taken from the RTL representations of the functions in Listing 3.1.

With this level of semantic information, the RTL representation of the firmware image can be compared to determine function level semantic changes. When a function has been identified as changed, metadata about the function needs to be generated. This metadata defines the location and length of the funtion in the new firmware image, the location of the function in the old image and a set of relocations. Similar to relocations used by linkers and loaders, these

relocations provide the information that the upgrade mechanism needs to correctly execute the new function in a different environment. Specifically, there are four types of relocations that are used in the metadata:

- **Insert Value Relocation**: This relocation is used when the patched function P calls a function F that has not been changed semantically between the old and the new firmware image, but has shifted in memory. In this case the address of F in the old firmware image is inserted into the literal pool of P where new address of F is located. This can not be done before transmitting the patch to the satellite as the address of F in the new firmware image needs to be preserved for when the new image is executed from flash.
- **Rodata Relocation**: This relocation is needed to ensure constants are correctly accessed from the patched functions running in RAM. If the new firmware image was already written to flash, this would be a trivial case to handle. The address of the constant in the literal pool would only have to be updated to point to flash bank 2 instead of bank 1. In the default configuration this can be done by adding 0x100000 to the value in the literal pool which contains the address of the constant. However, to apply the patch as soon as possible the new firmware image is only written to flash bank 2 on demand. At first, only the patched functions are written to RAM, so the modified constants also need to be written to RAM. Then, their addresses are written to the corresponding literal pool entries of the patched function.
- **New Variable Relocation**: When a new global variable is introduced in the new firmware image, memory needs to be allocated for it in RAM. Also, the address of the newly allocated variable needs to be inserted into the literal pool of the patched functions that accesses it. The value of the new variable is initialized as zero.
- **New Function Relocation**: If the patched function calls a function that previously hadn't been in the firmware image, which might also be the case if the function was a previously unused library function and the linker discarded it, the function is also included in the metadata as a patch. The address of the patch is then inserted into the literal pool of the calling function.

After the relocations have been applied on the device, the breakpoints can be activated and the patched functions can be executed from RAM.

3.4 Device State Handling & Crash Consistency

To ensure that the upgrade mechanism does not open the door to inconsistencies in the state of the device, multiple measures are taken.

3.4.1 Persistent State

Most critical is the persistent state of the device which is defined by the contents of its flash memory as only these contents endure a power cycle. If this state is not well maintained, the device might not be able to boot after a power cycle which in the worst case means the end of the mission. Thus, it is important to ensure that the software upgrade mechanism reliably writes the new firmware image to flash memory. In the project at hand, the mechanism also needs to initiate the swap of flash banks to make the new firmware image active. This

results in a trade-off between maintaining the persistent state, not blocking the device for too long by time consuming flash write operations and not exceeding the power budget of the device.

To deal with this trade-off, the upgrade mechanism implements a flash write procedure that performs flash write operations in a non-blocking way. This is done by relying on the flash interrupt that the STM32H7 provides. Specifically, the HAL library provides functions to initiate flash erase and write operations and provides an interrupt handler that maintains the state of the flash peripheral which can be used to determine when the peripheral is ready for the next operation. To reduce the memory overhead of keeping large parts of the new firmware image in RAM, the flash write procedure only loads 1 kB of the new firmware image into RAM at a time. In the production environment the flash write procedure can be flexibly configured to load more or less of the new firmware image into RAM at a time. It can also be configured to pause the flash write procedure after any operation to save resources.

3.4.2 Debug infrastructure

The system to change the control flow of the program relies on core elements of the debug infrastructure of the STM32H7. The Debug Monitor can not only be used to handle breakpoints defined by the FPB, but also to handle other debug events. It needs to be ensured that the Debug Monitor exception handler does not show unexpected behavior if a debug event from one of these other sources occurs. The specific scenarios in which the Debug Monitor exception handler can be triggered are:

- A debug event is generated by a breakpoint instruction
- The application sets the `DEMCR.MON_PEND` or the `DEMCR.MON_STEP` bit which makes the device raise a debug monitor exception.
- One of the Data Watchpoint and Trace Unit (DWT)'s watchpoints was triggered.
- An external debug request was received by the CPU via the `EDBGRQ` signal.

The custom Debug Monitor exception handler that is part of the upgrade mechanism implements safety measures to deal with these scenarios. If it encounters a breakpoint not present in the patch table maintained on the device, it removes the breakpoints from the FPB and returns to the address from which it was called. When it encounters a breakpoint instruction it sets the program counter to the address after the breakpoint instruction and continues execution. Further debug events can be handled by setting the appropriate bit in the Cortex-M7's `DFSR` register. This register is used to indicate the source of the debug event. By writing to this register, the Debug Monitor signals the system that it has handled the debug event and that the program can continue execution.

Any of the above mentioned events are highly unlikely. However, as they are not impossible, the Debug Monitor exception handler needs to be able to handle them gracefully.

4 Evaluation

The following sections present the evaluation of the firmware upgrade mechanism based on the key requirements to it. The first section evaluates the uplink traffic reduction achieved by the patching system as well as its decompression performance. It is followed by the evaluation of the performance of the patch execution environment, including the overhead caused by the Debug Monitor exception handler. In the final sections, the semantic change analysis and the device state handling are evaluated.

4.1 Traffic Reduction during Dissemination

With respect to disseminating the software upgrade, the primary target is to create the smallest possible artifact that can be decompressed with minimal resource requirements. This section will present results of a comparison between different diff and compression algorithms to identify an optimal combination for this purpose. Furthermore, it will give reference for how resource requirements on the STM32H7 change with the size of the artifact.

4.1.1 Minimizing the Compression Ratio

Three diff algorithms and four compression algorithms were selected to evaluate which combination of diff and compression algorithms performs best. The diff algorithms are xdelta3, bsdiff, HDiffPatch. The compression algorithms are LZMA, Bzip2, zstd and PPM.

Methodology

The evaluation was performed using three repositories of open source software for embedded systems as well as a repository used for testing the proposed patching system. The repositories were selected to represent different types of software. One repository, strawberryhacker/fat32, is a library to interact with FAT32 filesystems[Str25]. Another, symisc/vedis, is a lightweight equivalent of the Redis in-memory database for embedded systems[Sys21]. The third repository, marcinbor85/cAT, is a C library to parse standard and custom AT commands on embedded systems[Bor21]. For each of these repositories, two positions in the git history were selected, between which the difference in the source code was similar to what can be expected in a typical patch scenario. In all three cases, there were changes in around 10 code positions, with the total number of changed lines being around 40. Each version of each repository was compiled with the same compiler and compiler options as the image used for testing the patching system. The pairs of images were then used to create the diffs and compress them using the selected algorithms.

Results

The results of the evaluation are presented in Tables 4.1 and 4.2. What stands out is that the artifact created by the HDiffPatch diff algorithm is significantly larger than the artifacts created

Sample	New Image Size	HDiffPatch	xdelta3	bsdiff
FAT32	26536	91.0%	16.9%	7.9%
Vedis	110124	90.5%	12.1%	4.1%
cAT	42404	88.6%	18.1%	11.7%
S3P	79804	100.0%	3.5%	1.6%

Table 4.1: Diff performance: compression ratios for different patching tools. The compression ratio in this case is calculated based on the size of the new image based on the reference by Sun et al [Sun+24]. A lower compression ratio indicates better performance. S3P is the repository used for testing the patching system.

Algorithm	xdelta3				HDiffPatch				bsdiff			
	FAT32	cAT	Vedis	**AVG**	FAT32	cAT	Vedis	**AVG**	FAT32	cAT	Vedis	**AVG**
LZMA	16.9%	18.3%	12.2%	**15.8%**	6.4%	7.2%	4.0%	**5.9%**	7.9%	11.7%	4.1%	**7.9%**
zstd	16.7%	18.0%	12.1%	**15.6%**	6.5%	7.3%	4.0%	**5.9%**	7.8%	11.6%	4.1%	**7.8%**
PPM	18.0%	19.0%	12.5%	**16.5%**	6.4%	7.3%	3.6%	**5.8%**	8.7%	12.3%	4.3%	**8.4%**
Bzip2	18.5%	19.2%	12.5%	**16.7%**	9.3%	12.7%	4.5%	**8.8%**	7.5%	8.5%	4.2%	**6.7%**

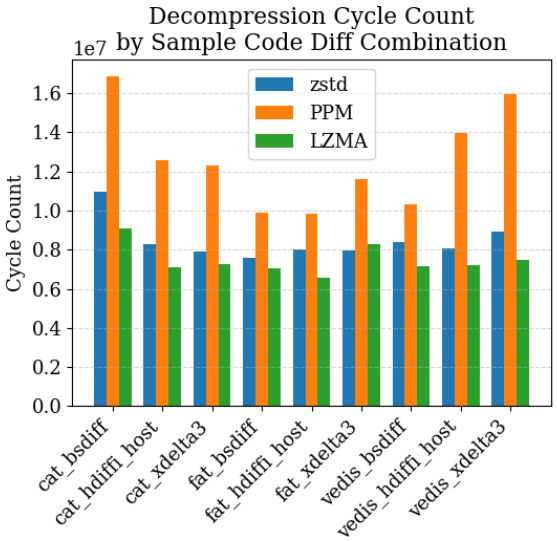
Table 4.2: Compression performance of different diff formats with various compressors. A lower compression ratio indicates better performance. In this table the results from the patching system are excluded as Table 4.1 suggests strong difference between naturally created patches and artificially created patches in the patching system. The averages across the three repositories are shown in the last column for each method. The best combination of diff and compression algorithm is highlighted in green.

by the other two algorithms. Its size is even in the same order of magnitude as the size of the original image. However, Table 4.2 reveals that the artifact created by HDiffPatch can be compressed to a significantly smaller size than the artifacts created by the other two algorithms. Especially the artifact created by xdelta3 does not compress well. There is also a strong contrast between the performance of each of the algorithms for the naturally created patches in Fat32, Vedis and cAT and the performance of the algorithms for the patches created as examples in the patching system (S3P). While HDiffPatch performs significantly worse with the artificially created patches, xdelta3 and bsdiff perform significantly better with them.

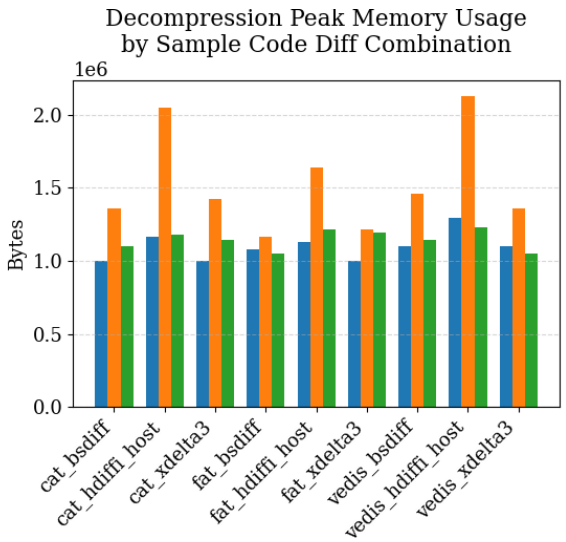
Table 4.2 also shows that from the algorithms selected for the evaluation, LZMA, zstd and PPM perform at a nearly equal level. Bzip2 on the other hand does not perform as well. Both findings are in line with the research previously cited. Further investigation into the compression performance of the algorithms revealed that the compression ratios can only be minimally improved by changing the parameters of the algorithms. Though for both algorithms this resulted in an artifact 1-2% smaller, the difference in compression ratio is below 0.1%. The Appendix includes details on these measurements.

4.1.2 Decompression Performance

Given the results of the previous subsection, the next step is to evaluate the decompression performance of the artifacts created by the best performing combinations of diff and compression algorithms. The combinations selected were HDiffPatch with LZMA, zstd and PPM.



(a) CPU cycles required for decompression



(b) Peak memory usage during decompression

Figure 4.1: Decompression performance of the artifacts created by HDiffPatch with LZMA, zstd and PPM on the host computer. The left figure shows the number of CPU cycles required for decompression, while the right figure shows the peak memory usage during decompression.
The x-axes show the combinations of code sample, e.g. FAT32, cAT and Vedis, combined with the diff algorithm used, e.g. HDiffPatch, xdelta3 and bsdiff.

Methodology

To reduce the complexity of the evaluation, the decompression performance was evaluated not on the target device, the STM32H7, but on a host computer. The host computer is a 2020 MacBook Pro with an Apple M1 chip and 8 GB of RAM. The endpoints of the experiment were the number of CPU cycles required and the maximum memory usage during decompression. The data used were the artifacts of the three repositories FAT32, Vedis and cAT with which the HDiffPatch algorithm was used. The decompression performance was measured using the `time` tool on MacOS.

Results

Figure 4.1 reveals the results of the decompression performance evaluation. It shows that the resource consumption of LZMA and zstd is comparable both in respsect to the maximum memory usage and the number of cycles executed. PPM on the other hand consumes significantly more memory and requires significantly more CPU cycles. For the CPU cycles required this is especially noticeable for the combination of cAT and bsdiff and for the combination of Vedis and xdelta3. However, for the peak memory usage PPM stands out negatively for any artifacts created with HDiffPatch. For each of these artifacts the compression ratio achieved by PPM is significantly higher than for the others. Meanwhile zstd and LZMA show significantly less variation in their resource requirements.

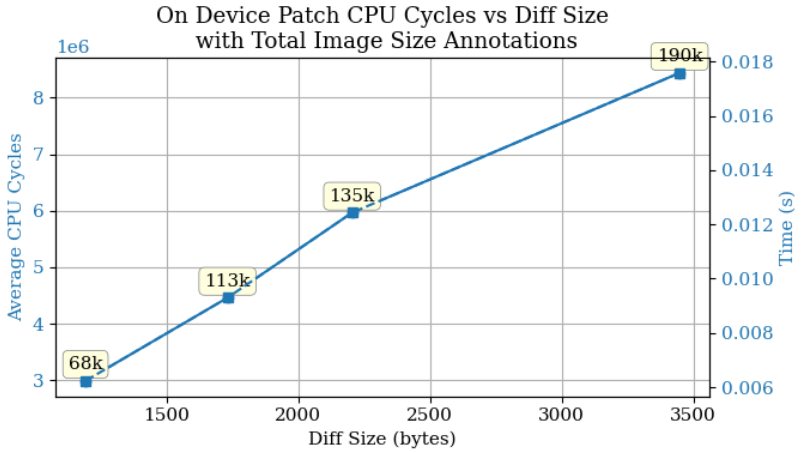


Figure 4.2: On device decompression performance of variable size firmware images resulting in variable size patches. The x-axis shows the size of the patch in bytes, while the y-axes show CPU cycles and the equivalent time in seconds at 480 MHz. The data points are the average of 5 measurements for each firmware image size.

4.1.3 On Device Decompression Performance

Methodology

To determine the decompression performance on the target device and how it changes with the size of the artifact, the combination of HDiffPatch and LZMA was selected. Four firmware images of 68 kB, 113 kB, 135 kB and 190 kB were created by adding library code to the existing firmware image used for testing the patching system. Patches and metadata were created for each of these images. Then, the cycle count was measured starting from when the full patch artifact was received via Universal Asynchronous Receiver/Transmitter (UART) to when the functions that were changed by the patch were loaded into RAM and the FPB was configured to redirect control flow to the patched code.

Results

The results of the decompression performance on the target device are presented in figure 4.2. They present a linear relationship between the size of the patch and the cycle count required for decompression and patch application.

4.1.4 Discussion

The results of the evaluation of different diff and compression algorithms give clear guidance for the project at hand. While HDiffPatch performs poorly without compression, it outperforms the other two algorithms significantly considering compression. If it is combined with LZMA or zstd, it also performs well in terms of decompression performance. Further research needs to show whether zstd's training features creates a significant advantage over LZMA in terms of

compression ratio. As long as no clear advantage by zstd can be shown, LZMA is the better choice as it integrates better with HDiffPatch.
It also needs to be mentioned that because zstd features a reverse decoding sequence[Fac25a], meaning that within each compression block the decompressed data becomes available in reverse order. This would require extra effort in combining it efficiently with HDiffPatch as it does not decompress the entire patch first before applying it, but applies the patch while decompressing it piecewise.
The investigation of the decompression and patch application performance on the target device shows that even with growing image sizes, the resources required for this step do not grow excessively. It is likely that the target system will have to interrupt the decompression and patch application to execute other tasks with higher priority. This can be facilitated by the use of an RTOS.

4.2 Upgrade Execution Environment

This section will evaluate the performance of the patch system in the target environment. Specifically, it will present the overhead caused by running the patched code from RAM compared to running it from flash memory.

4.2.1 Methodology

To measure the amount of CPU cycles used to execute instructions, the STM32H7 features a cycle counter as part of the DWT. Once activated, the cycle counter increments a 32bit register with each clock cycle. Assuming that the STM32H7 runs at its maximum clock speed of 480 MHz, this register will overflow after approximately 9 seconds. This is sufficient for the measurements in this section. To identify the overhead caused by running the patched code from RAM, the cycle counter is reset before executing one of the patched functions. Then, it is read at the very end of the Debug Monitor exception handler before execution jumps to the patched code in RAM. After execution returns from the patched code, the cycle counter is read again. This setup allows to both measure the performance of the Debug Monitor as well as identify any overhead caused by running the patched code from RAM. The measurements were performed as two independent test runs of the same firmware image. One test run featured ten repititions of all patched functions executed from RAM and afterwards from flash. The second test run featured 20 repetitions in the same setting. Compiler optimization for size using `-Os` was enabled to ensure that the test scenario is as close to a real world scenario as possible.

4.2.2 Results

The results of the debug monitor overhead measurements are presented in figure 4.3. The figure shows the average overhead caused by the Debug Monitor exception handler for each of the patched functions. Looking at the exact values, the cycle overhead can be stated as between 127 cycles and 252 cycles. The results for the Debug Monitor overhead from the two test runs are consistent. There is no significant difference between the cycle counts achieved across the test runs. Only within each test run the cycle count for the first two invocations of the patched functions is generally higher.
For the overhead caused by running the patched code from RAM, the results are presented in figure 4.4. There is no clear tendency towards an overhead visible in the results. All but one function perform consistently within and across the test runs and show little variance

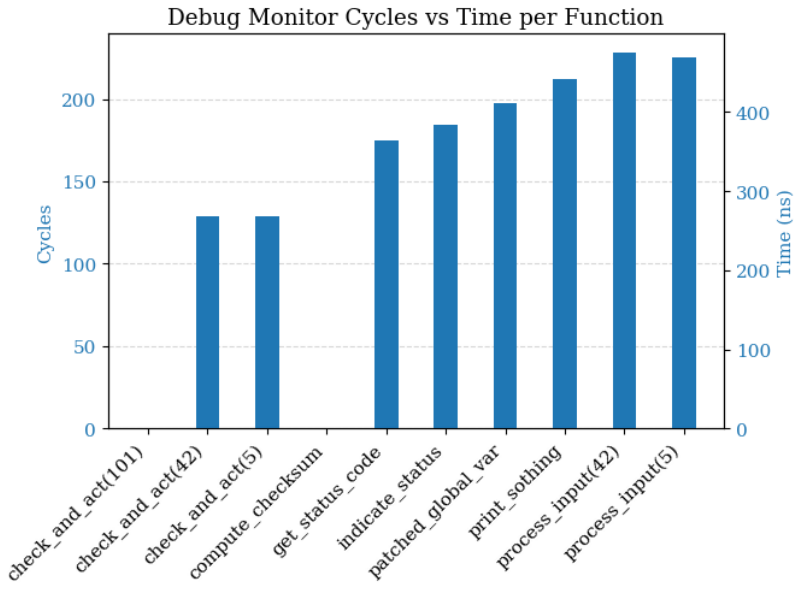


Figure 4.3: Cycle overhead caused by the Debug Monitor exception handler for each of the patched functions. The order of the functions in the x-axis is the order in which they are present in the patch table maintained on the device. The y-axis shows the cycle overhead in cycles and the equivalent time in seconds at 480 MHz. The data points are the average of 20 measurements for each function.

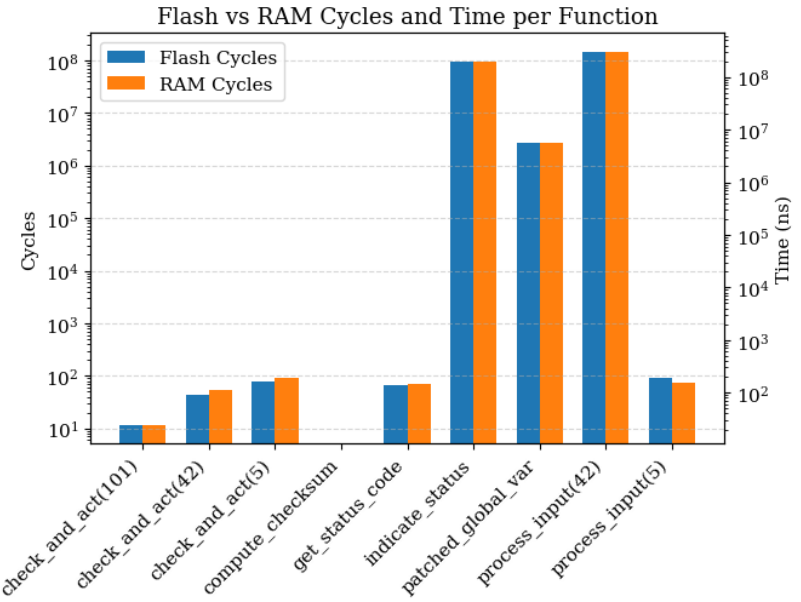


Figure 4.4: Cycle count of the patched functions executed from RAM and flash. The y-axis shows the cycle overhead in cycles and the equivalent time in seconds at 480 MHz.

in their execution times. Specifically, the standard deviation relative to the mean of cycle times varies between the functions but only in one case exceeds 2%. An overhead between 2 and 11 cycles caused by running the patched code can be seen in four settings, namely `check_and_act(42)`, `check_and_act(5)`, `get_status_code` and `patched_global_var`. In two settings, namely `process_input` and `indicate_status`, the cycle count is lower when running the patched code from RAM than when running it from flash. For `process_input(42)` there is an overhead in one test run, but not in the other.

4.2.3 Discussion

There is a clear tendency in the overhead caused by the Debug Monitor exception. It increases with the index of the function in the patch table maintained on the device. This order depends on the order in which the patch metadata is transmitted. In the test scenario, the order was `check_and_act`, `compute_checksum`, `get_status_code`, `indicate_status`, `patched_global_var`, `print_something`, `process_input`, `HAL_Delay`. This overhead is well explainable, as the Debug Monitor exception handler has to find the right entry in the patch table by the address where the exception was triggered. This lookup could be optimized using a hash table or a binary search tree, but the performance gain would be negligable, as the difference between the maximum and minimum execution time of the Debug Monitor amounts to 200 ns for the device running at 480 MHz.
The increased overhead for the first two invocations of the patched functions in each test run can be explained by caching. The STM32H7 features a Data Cache (DCache) and an Instruction Cache (ICache) which were both enabled for the test runs. During the first invocations of a patched function, the instructions and data are loaded into the ICache and DCache respectively. In the later invocations, the cached versions are used, resulting in lower overhead.

A significant overhead caused by running the patched functions from RAM could not be observed. In most cases the overhead is less than 10 cycles, which is equivalent to 20 ns at 480 MHz. However, the results in this category were not as clear as the results for the Debug Monitor overhead. Further investigation is needed to determine the reason why two functions persistently show a lower cycle count when executed from RAM. Also, the reason why one function shows a lower cycle count in one test run and a higher cycle count in the other is not clear. So far, it only stands out that in two of these cases the `HAL_Delay` function is called, which is used to implement a delay in milliseconds. All other functions with which the experiment was performed do not use this function. `HAL_Delay` is certainly expected to introduce variance in the results. It is not clear however why some functions that make use of it show consistent behaviour and others don't.

The performance of the Debug Monitor seen in this evaluation is in line with the performance achieved by Yi et al.'s RapidPatch. Yi et al. report overheads of 135 and 252 cycles for the Debug Monitor based implementation of their patching system[He+22]. Yi et al. also present results for the overhead caused by code instrumentation. They show that the overhead caused by jumping to the patch code might be lower with 66 cycles, but this overhead is also encountered at all code positions for which no patch exists.

The results regarding executing the patch from RAM are consistent with the results obtained by Niesler et al. While their patch system made use of the FPB to redirect control flow, they also executed the patched code directly from RAM and didn't observe a significant overhead[NSD21]. It needs to be noted, however, that their experiments were conducted on a different device, the STM32F4 and used patches for RTOSs which likely didn't include calls to `HAL_Delay`.

Executing the patch from RAM has some clear advantages of the proposed system over the RapidPatch system. Yi et al. report that calling a C function from their eBPF runtime causes an overhead of 300 cycles. This is clearly not a problem in the system at hand, as the example function get_status_code calls two C functions and runs in less than 100 cycles from RAM. Salehi and Pattabiraman underline this point by showing that their patching system, AutoPatch, has a performance advantage over RapidPatch of 50% in terms of patch execution time[SP24]. Though not stated explicitly that they execute instructions from RAM, they stress that in their system patches are disseminated as binary, executable functions which suggests that they are either executed from RAM or flash.

4.3 Semantic Change Analysis

Given that a significant part of the implementation of this project was the semantic analyser in charge of determining functions that changed, this system also needs to be evaluated.
Based on limitations of other patch systems such as AutoPatch, one clear limitation was found: So far, it is not possible to handle changes in global C struct variables. If the new image adds a member to the struct, the size of the global variable in the .data or .bss section has to change. By default, this can not be done at runtime as other global variables would have to move. As a consequence, code that accesses global variables that are stored at a higher memory address as the changed struct would have to be changed. Given that instructions are stored in flash, this would cause a massive overhead.
Furthermore, there is a caveat related to literal pools in long functions. The reason for this is that the program counter relative load operation used in ARM's 16 bit Thumb instruction set has only a range of 1024 bytes. If the function is longer than that, and an instruction at the very beginning of the function needs to load a constant, a literal pool is created inside the function and not at the end. To apply relocations to such a literal pool, the offset of the literal pool from the start of the function needs to be known. For this, it needs to be known for each instruction between the start of the functions and the literal pool whether it is a 32 bit instruction or a 16 bit instruction. A simple solution to this problem is to build a map between the identifier GCC uses in RTL for an instruction and the instruction's size. This however is future work.

4.4 Device State Handling & Crash Consistency

This section presents the evaluation of the upgrade mechanisms persistent state management. Specifically, the time and resources consumed by the device when writing a new firmware image to flash memory was investigated.

4.4.1 Methodology

The experimental setup included an old and a new firmware image of both 70 kB and a patch of 1476 B including metadata. The flash write system was modified so that the cycle counter was activated when a new task of the flash write system was started. The cycle counter was read after the task was finished. Furthermore, when the interrupt handler triggered by the flash interrupt was entered, the cycle counter was also started. At the end of the interrupt handler the cycle counter was read again. Whenever the cycle counter was read, the value was used to increment a global counter of cycles executed by the flash write system. For each operation the cycle counts and the operation type (erase, load or program) were transmitted to the host computer

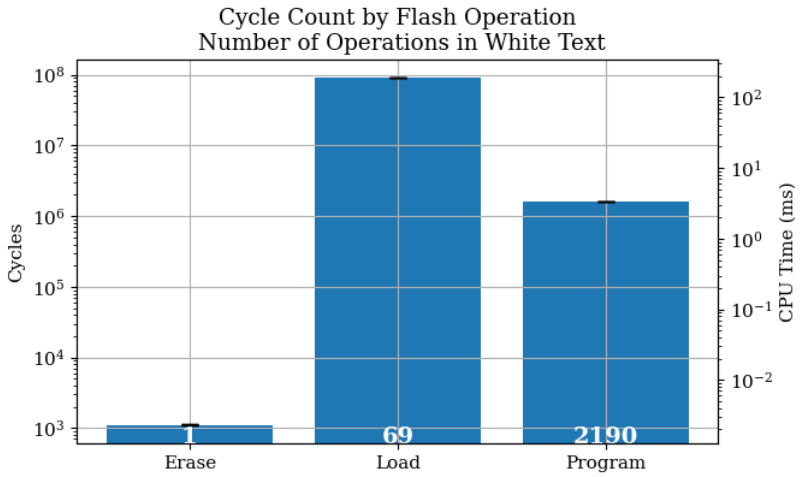


Figure 4.5: Cycle count and CPU time by flash operation. The CPU time is calculated based on a clock frequency of 480 MHz.
Erase and Program operations are not blocking, but Load operations are. The total flash write procedure took 23.4 s on average, as Program and Erase operations were performed in the background.

via UART. As the overall time for writing the new firmware image exceeds the overflow time of the cycle counter, this time was measured by the arrival times of the first and last cycle count message. For the load operations that extract sets of 32 byte flash words from the compressed new image, a set size of 32 was selected. This way, the system occupies 1024 B of RAM for buffering chunks of the new image before writing them to flash.

4.4.2 Results

The results of the evaluation are presented in figure 4.5. The figure shows that the primary source of computational overhead in the flash write system is caused by loading new flash words from the compressed image into RAM. The erase operation that erases the whole second flash bank consumes the least cycles and has a negligable impact. The impact of the flash program operations is also not significant as it is 1.7 % of the total cycles consumed. Furthermore, the experiment revealed the granularity of the computational overhead caused by each of the operations. As also presented in figure 4.5, the load operations have the lowest granularity, meaning there are relatively few, relatively large load operations. The overall image write operation required 69 load operations, which results in an average cycle count of 1337174 cycles per load operation. Given the maximum clock speed of the STM32H7 of 480 MHz, this is equivalent to 2.78 ms of CPU time per load operation. The erase and program operations on the other hand have a significantly lower average cycle count of 1101 and 2190 cycles respectively. The overall time for the flash write procedure was on average 23.4 s. Based on the cycle counts, the device was busy executing the flash write system for 0.196 s or 0.84% of this time.

4.4.3 Discussion

The findings from this experiment are in line with the expectations. Flash operations are known to be slow, but a non-blocking implementation using interrupts allows the device to continue executing other tasks while a new image is written to flash. The load operations are clearly the most expensive operation in terms of cycles consumed. It has been shown that they consume

a lot of cycles in total, but also per operation. This is very undesirable, as the load operations may block other tasks from being executed in time. In a production system, that uses an RTOS, the impact of these load operations could be mitigated by given the flash write system a lower priority than other tasks. Then, the RTOS's scheduler would ensure that the load operations do not block the CPU when time critical tasks need to be executed. Combined with the fact the device was only busy executing the flash write system for 0.84% of the whole time, this is a promising result. It shows that the new firmware image can be written to flash soon after the patch is received. This ensures that the device is protected from falling back to an old firmware image in case of a crash. It is only vulnerable to such an incident during the time the new firmware image is being written to flash.
Such a case is a possible future research direction. Also, a system should be developed that allows the device to check whether it booted from the flash bank with the newest firmware image or not. Otherwise, the device might boot from an old firmware image if a crash occurs after the new firmware image was written to flash, but before the flash banks were swapped. This can easily be implemented by storing a version number at a fixed address in any firmware image. The version numbers on both flash banks could be read at boot time and compared.

5 Conclusion

This thesis explored the challenges and design considerations involved in building an end-to-end firmware live-patching framework for a STM32H753-based CubeSat. It presented an upgrade mechanism that is particularly optimized for low bandwidth, low memory resources, fast patch application, and low overhead during patch execution. The details of how this optimization was implemented were discussed. This included a thorough analysis of diff and compression algorithms, of the STM32H7's Debug Monitor, and of the STM32H7's dual-bank flash architecture. Furthermore, the efforts in implementing a semantic analyser that minimizes limitations on which types of changes in the firmware can be applied were highlighted.

The upgrade mechanism was evaluated in terms of its compression ratio, which revealed the combination of HDiffPatch and LZMA to be the most effective and usable for the STM32H753. The evaluation also showed that this combination achieves a competitive decompression speed which scales linearly with the patch size.
The patch execution environment was shown to cause a maximum overhead of 0.53 µs through the Debug Monitor handling the control flow change to the patch for a device running at 480 MHz. Lastly, the flash write procedure was shown to reduce the CPU time usually associated with writing to flash memory from multiple seconds to less than 200 ms for a patch of 1.5 kB.

Further work could focus on further improving the compression ratio for the patch created by HDiffPatch. Especially zstd's training feature could not be evaluated in this work but might offer further improvements to the compression ratio. Similarly, it should be investigated whether the dictionaries created by LZMA show some similarity across different patches which are based on the same original firmware image. Ultimately, it might be beneficial to include a dictionary in the firmware image with which the CubeSat is launched. When a new patch is created and compressed, only extensions to this dictionary would have to be sent to the device. This might further reduce the compressed patch size.
Another area of research would be to efficiently store the state of the LZMA decompressor during decompression. This could significantly reduce the CPU time required for load operations during the flash write procedure.

The semantic analyser could be extended to not only support changes on the function level but also on the basic block level. Especially in case of large functions with very simple modifications this could significantly reduce the RAM requirements of the execution environment as not the whole function would have to be loaded into RAM.
Furthermore, the semantic analyser could be extended by a feature that traces the call hierarchy of interrupt handlers with higher execution priority than the Debug Monitor exception handler. This could help identify whether a patch would be called by such a high priority interrupt handler. Such a situation is highly important to avoid as the device raises a HardFault when the Debug Monitor is active and a breakpoint is hit from an interrupt handler with higher priority than the Debug Monitor exception handler. So far, the programmer or satellite operator has to ensure that this situation does not occur.

A Appendix

The firmware upgrade mechanism's source code can be found at `https://collaborating.tuhh.de/e-exk3/theses/ba_fredrik_wilke`.

A.1 Full Results of Compression Performance Evaluation

The following table shows the full results of the compression performance evaluation. It includes the results for test runs with different parameters for the best performing compression algorithms. The Deflate algorithm is represented by the zlib implementation.

Table A.1: Diff sizes for different combinations of patching and compression algorithms and their optimonal parameters.

Diff-Algo	Comp-Algo	Comp-Params	Name	Old Size	New Size	Diff Size
HDiffPatch	/	/	FAT32	26400	26536	24148
HDiffPatch	/	/	Vedis	109988	110124	99616
HDiffPatch	/	/	cAT	42164	42404	37552
HDiffPatch	/	/	S3P	79668	79804	79799
xdelta3	/	/	FAT32	26400	26536	4472
xdelta3	/	/	Vedis	109988	110124	13340
xdelta3	/	/	cAT	42164	42404	7671
xdelta3	/	/	S3P	79668	79804	2817
bsdiff	/	/	FAT32	26400	26536	2088
bsdiff	/	/	Vedis	109988	110124	4548
bsdiff	/	/	cAT	42164	42404	4961
bsdiff	/	/	S3P	79668	79804	1298
xdelta3	lzma	/	FAT32	26400	26536	4497
xdelta3	lzma	/	Vedis	109988	110124	13485
xdelta3	lzma	/	cAT	42164	42404	7739
xdelta3	lzma	/	S3P	79668	79804	2813
bsdiff	lzma	/	FAT32	26400	26536	2096
bsdiff	lzma	/	Vedis	109988	110124	4494
bsdiff	lzma	/	cAT	42164	42404	4956
bsdiff	lzma	/	S3P	79668	79804	1276
HDiffPatch	lzma	/	FAT32	26400	26536	1698
HDiffPatch	lzma	/	Vedis	109988	110124	4437
HDiffPatch	lzma	/	cAT	42164	42404	3070
HDiffPatch	lzma	/	S3P	79668	79804	1243
xdelta3	bzip2	/	FAT32	26400	26536	4915
xdelta3	bzip2	/	Vedis	109988	110124	13808
xdelta3	bzip2	/	cAT	42164	42404	8132
xdelta3	bzip2	/	S3P	79668	79804	3238
bsdiff	bzip2	/	FAT32	26400	26536	2467
bsdiff	bzip2	/	Vedis	109988	110124	4944
bsdiff	bzip2	/	cAT	42164	42404	5403
bsdiff	bzip2	/	S3P	79668	79804	1621
HDiffPatch	bzip2	/	FAT32	26400	26536	1999
HDiffPatch	bzip2	/	Vedis	109988	110124	4621
HDiffPatch	bzip2	/	cAT	42164	42404	3587

Diff-Algo	Comp-Algo	Comp-Params	Name	Old Size	New Size	Diff Size
HDiffPatch	bzip2	/	S3P	79668	79804	1284
xdelta3	zstd	/	FAT32	26400	26536	4444
xdelta3	zstd	/	Vedis	109988	110124	13354
xdelta3	zstd	/	cAT	42164	42404	7646
xdelta3	zstd	/	S3P	79668	79804	2770
bsdiff	zstd	/	FAT32	26400	26536	2079
bsdiff	zstd	/	Vedis	109988	110124	4474
bsdiff	zstd	/	cAT	42164	42404	4923
bsdiff	zstd	/	S3P	79668	79804	1265
HDiffPatch	zstd	/	FAT32	26400	26536	1715
HDiffPatch	zstd	/	Vedis	109988	110124	4460
HDiffPatch	zstd	/	cAT	42164	42404	3089
HDiffPatch	zstd	/	S3P	79668	79804	1237
xdelta3	zstd	–ultra -22	FAT32	26400	26536	4444
xdelta3	zstd	–ultra -22	Vedis	109988	110124	13354
xdelta3	zstd	–ultra -22	cAT	42164	42404	7646
xdelta3	zstd	–ultra -22	S3P	79668	79804	2770
bsdiff	zstd	–ultra -22	FAT32	26400	26536	2079
bsdiff	zstd	–ultra -22	Vedis	109988	110124	4474
bsdiff	zstd	–ultra -22	cAT	42164	42404	4923
bsdiff	zstd	–ultra -22	S3P	79668	79804	1265
HDiffPatch	zstd	–ultra -22	FAT32	26400	26536	1703
HDiffPatch	zstd	–ultra -22	Vedis	109988	110124	4370
HDiffPatch	zstd	–ultra -22	cAT	42164	42404	3077
HDiffPatch	zstd	–ultra -22	S3P	79668	79804	1237
HDiffPatch	tuz	/	FAT32	26400	26536	1753
HDiffPatch	tuz	/	Vedis	109988	110124	4736
HDiffPatch	tuz	/	cAT	42164	42404	3292
HDiffPatch	tuz	/	S3P	79668	79804	
HDiffPatch	tuzi	/	FAT32	26400	26536	1766
HDiffPatch	tuzi	/	Vedis	109988	110124	4749
HDiffPatch	tuzi	/	cAT	42164	42404	3299
HDiffPatch	tuzi	/	S3P	79668	79804	
HDiffPatch	zlib	/	FAT32	26400	26536	1796
HDiffPatch	zlib	/	Vedis	109988	110124	4828
HDiffPatch	zlib	/	cAT	42164	42404	3351
HDiffPatch	zlib	/	S3P	79668	79804	
xdelta3	ppmd	-mx=9	FAT32	26400	26536	4775
xdelta3	ppmd	-mx=9	Vedis	109988	110124	13806
xdelta3	ppmd	-mx=9	cAT	42164	42404	8044
xdelta3	ppmd	-mx=9	S3P	79668	79804	3067
bsdiff	ppmd	-mx=9	FAT32	26400	26536	2314
bsdiff	ppmd	-mx=9	Vedis	109988	110124	4734
bsdiff	ppmd	-mx=9	cAT	42164	42404	5211
bsdiff	ppmd	-mx=9	S3P	79668	79804	1474
HDiffPatch	ppmd	-mx=9	FAT32	26400	26536	1693
HDiffPatch	ppmd	-mx=9	Vedis	109988	110124	3991
HDiffPatch	ppmd	-mx=9	cAT	42164	42404	3100
HDiffPatch	ppmd	-mx=9	S3P	79668	79804	1125
HDiffPatch	lzma	level 1	FAT32	26400	26536	1836
HDiffPatch	lzma	level 1	Vedis	109988	110124	4876
HDiffPatch	lzma	level 1	cAT	42164	42404	3355
HDiffPatch	lzma	level 1	S3P	79668	79804	
HDiffPatch	lzma	level 4	FAT32	26400	26536	1836
HDiffPatch	lzma	level 4	Vedis	109988	110124	4876
HDiffPatch	lzma	level 4	cAT	42164	42404	3355

Diff-Algo	Comp-Algo	Comp-Params	Name	Old Size	New Size	Diff Size
HDiffPatch	lzma	level 4	S3P	79668	79804	
HDiffPatch	lzma	level 9	FAT32	26400	26536	1698
HDiffPatch	lzma	level 9	Vedis	109988	110124	4437
HDiffPatch	lzma	level 9	cAT	42164	42404	3070
HDiffPatch	lzma	level 9	S3P	79668	79804	
HDiffPatch	lzma	dict-size 64k	FAT32	26400	26536	1698
HDiffPatch	lzma	dict-size 64k	Vedis	109988	110124	4463
HDiffPatch	lzma	dict-size 64k	cAT	42164	42404	3070
HDiffPatch	lzma	dict-size 64k	S3P	79668	79804	
HDiffPatch	lzma	dict-size 64k level 9	FAT32	26400	26536	1698
HDiffPatch	lzma	dict-size 64k level 9	Vedis	109988	110124	4463
HDiffPatch	lzma	dict-size 64k level 9	cAT	42164	42404	3070
HDiffPatch	lzma	dict-size 64k level 9	S3P	79668	79804	
HDiffPatch	lzma	dict-size 16k	FAT32	26400	26536	1696
HDiffPatch	lzma	dict-size 16k	Vedis	109988	110124	4432
HDiffPatch	lzma	dict-size 16k	cAT	42164	42404	3066
HDiffPatch	lzma	dict-size 16k	S3P	79668	79804	
HDiffPatch	lzma	dict-size 8k	FAT32	26400	26536	1692
HDiffPatch	lzma	dict-size 8k	Vedis	109988	110124	4423
HDiffPatch	lzma	dict-size 8k	cAT	42164	42404	3067
HDiffPatch	lzma	dict-size 8k	S3P	79668	79804	
HDiffPatch	lzma	dict-size 4k	FAT32	26400	26536	1687
HDiffPatch	lzma	dict-size 4k	Vedis	109988	110124	4434
HDiffPatch	lzma	dict-size 4k	cAT	42164	42404	3066
HDiffPatch	lzma	dict-size 4k	S3P	79668	79804	

Bibliography

[Ara+21] Konstantinos Arakadakis et al. “Firmware Over-the-air Programming Techniques for IoT Networks - A Survey”. In: *ACM Comput. Surv.* 54.9 (Oct. 2021). ISSN: 0360-0300. DOI: 10.1145/3472292. URL: https://doi.org/10.1145/3472292.

[Bor21] Marcin Borowicz. *libcat*. Accessed: 2025-07-17. 2021. URL: https://github.com/marcinbor85/cAT.

[BS13] Stephen Brown and Cormac J. Sreenan. “Software Updating in Wireless Sensor Networks: A Survey and Lacunae”. In: *Journal of Sensor and Actuator Networks* 2.4 (2013), pp. 717–760. ISSN: 2224-2708. DOI: 10.3390/jsan2040717. URL: https://www.mdpi.com/2224-2708/2/4/717.

[CKM17] César Coelho, Otto Koudelka, and Mario Merri. “NanoSat MO framework: When OBSW turns into apps”. In: *2017 IEEE Aerospace Conference*. IEEE. 2017, pp. 1–8.

[End21] EnduroSat. *EnduroSat Products Presentation*. https://www.endurosat.com/wp-content/uploads/2021/08/EnduroSat_Products_Presentation_08_2021.pdf. Accessed July 2025. 2021.

[End24] EnduroSat AD. *UHF Transceiver Type II User Manual*. Version: 4.14. 2024.

[Fac25a] Facebook. *Zstandard compression format*. Accessed: 2025-07-16. 2025. URL: https://github.com/facebook/zstd/blob/dev/doc/zstd_compression_format.md#entropy-encoding.

[Fac25b] Facebook. *Zstandard compression format*. Accessed: 2025-07-17. 2025. URL: https://github.com/facebook/zstd.

[GBK17] Apoorv Gupta, Aman Bansal, and Vidhi Khanduja. “Modern lossless compression techniques: Review, comparison and analysis”. In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. 2017, pp. 1–8. DOI: 10.1109/ICECCT.2017.8117850.

[GNU24] GNU. *patch(1) — Linux manual page*. Accessed: 2025-07-06. 2024. URL: https://man7.org/linux/man-pages/man1/patch.1.html.

[Gum25] GumTreeDiff. *GumTree*. Accessed: 2025-07-27. 2025. URL: \url{https://github.com/GumTreeDiff/gumtree}.

[He+22] Yi He et al. “{RapidPatch}: firmware hotpatching for {Real-Time} embedded devices”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2225–2242.

[KBM19] Ondrej Kachman, Marcel Balaz, and Peter Malik. “Universal framework for remote firmware updates of low-power devices”. In: *Computer Communications* 139 (2019), pp. 91–102. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2019.03.014. URL: https://www.sciencedirect.com/science/article/pii/S0140366418307722.

[Kor+02] David Korn et al. *The VCDIFF Generic Differencing and Compression Data Format*. RFC 3284. July 2002. DOI: 10.17487/RFC3284. URL: https://www.rfc-editor.org/info/rfc3284.

[Li+07] Weijia Li et al. "UCC: update-conscious compilation for energy efficiency in wireless sensor networks". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 383–393.

[lib] libcsp. *The Cubesat Space Protocol*. Accessed: 2025-07-24. URL: https://github.com/libcsp/libcsp.

[Lima] ARM Limited. *Arm GNU Toolchain*. Accessed: 2025-07-14. URL: https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain.

[Limb] ARM Limited. *Arm Toolchain for Embedded 20.0.0 Open Beta Release Notes*. https://developer.arm.co
overview/Release-highlights?lang=en. Accessed July 2025.

[Lim09] ARM Limited. *Cortex-M4 Technical Reference Manual, Revision r0p0*. ARM Limited, 2009.

[Lu+20] Tao Lu et al. "Adaptively Compressing IoT Data on the Resource-constrained Edge". In: *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020. URL: https://www.usenix.org/conference/hotedge20/presentation/lu.

[Mac] Joshua P. MacDonald. *xdelta3*. https://github.com/jmacd/xdelta. Accessed July 2025.

[Mor+21] Brendan Moran et al. *A Firmware Update Architecture for Internet of Things*. RFC 9019. Apr. 2021. DOI: 10.17487/RFC9019. URL: https://www.rfc-editor.org/info/rfc9019.

[NSD21] Christian Niesler, Sebastian Surminski, and Lucas Davi. "HERA: Hotpatching of Embedded Real-time Applications." In: *NDSS*. 2021.

[Oro18] Ervin Oro. *Delta Updates for ESTCube-2 Onboard Computer Software*. 2018.

[Rov+23] Mikhail. M. Rovnyagin et al. "Data Exchange Acceleration Methods in a Decentralized File System". In: *2023 Seminar on Information Computing and Processing (ICP)*. 2023, pp. 174–180. DOI: 10.1109/ICP60417.2023.10397171.

[SM09] David Salomon and Giovanni Motta. *Handbook of Data Compression*. 5th. Springer Publishing Company, Incorporated, 2009. ISBN: 1848829027.

[Sona] Hou Si Song. *HDiffPatch*. https://github.com/sisong/HDiffPatch. Accessed July 2025.

[Sonb] Hou Si Song. *HPatchLite*. https://github.com/sisong/HPatchLite. Accessed July 2025.

[SP24] Mohsen Salehi and Karthik Pattabiraman. "AutoPatch: Automated Generation of Hotpatches for Real-Time Embedded Devices". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. ACM, Dec. 2024, 2370–2384. DOI: 10.1145/3658644.3690255. URL: http://dx.doi.org/10.1145/3658644.3690255.

[STM09] STMicroelectronics. *RM0433 Reference manual*. STMicroelectronics, 2009.

[STM23] STMicroelectronics. *STM32H753xI Datasheet DS12117 Rev 9*. STMicroelectronics, 2023.

[Str25] StrawberryHacker. *FAT32 File System*. Accessed: 2025-07-17. 2025. URL: https://github.com/strawberryhacker/fat32.

[Sun+24] Tong Sun et al. "Understanding Differencing Algorithms for Mobile Application Updates". In: *IEEE Transactions on Mobile Computing* 23.12 (2024), pp. 12146–12160. DOI: 10.1109/TMC.2024.3407867.

[Sys21] Symisc Systems. *Vedis Embedded Database*. Accessed: 2025-07-17. 2021. URL: https://github.com/symisc/vedis.

[Zan+22] Koen Zandberg et al. "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers". In: *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. Middleware '22. Quebec, QC, Canada: Association for Computing Machinery, 2022, 161–173. ISBN: 9781450393409. DOI: 10.1145/3528535.3565242. URL: https://doi.org/10.1145/3528535.3565242.

[Zho+24] Ming Zhou et al. "Save the Bruised Striver: A Reliable Live Patching Framework for Protecting Real-World PLCs". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. EuroSys '24. Athens, Greece: Association for Computing Machinery, 2024, 1192–1207. ISBN: 9798400704376. DOI: 10.1145/3627703.3650068. URL: https://doi.org/10.1145/3627703.3650068.