One Small Patch for a File, One Giant Leap for OTA Updates

Julian Rederlechner
CISPA Helmholtz Center for Information Security
julian.rederlechner@cispa.de

Ulysse Planta
CISPA Helmholtz Center for Information Security
ulysse.planta@cispa.de

Ali Abbasi
CISPA Helmholtz Center for Information Security
abbasi@cispa.de

***Abstract*—Over-the-Air (OTA) software updates are essential for satellite security and reliability, yet limited uplink bandwidth and communication windows make them challenging. To minimise data transfer, systems often use delta updates from binary diffing algorithms. While prior work has shown that the HDiffPatch algorithm outperforms many established diffing tools, update systems frequently rely on the bsdiff4 algorithm (e.g., through OSTree), and little is known about the suitability of its unpublished successors or their behaviour on space-relevant software. In this work, we perform the first comparative analysis of bsdiff4, bsdiff6, bsdiff-ra, and HDiffPatch on a dataset representative of satellite software stacks and payload data. Our results show that bsdiff6 produces, on average, ≈4,8% smaller patches than bsdiff4, outperforming it in 18 of 19 test cases, while providing stronger memory safety through its Rust-based implementation. On the other hand, HDiffPatch provides better results for compressed data. To enable this evaluation, we reconstruct a bsdiff6 implementation from original design notes, providing the first published version. In addition, a detailed analysis of bsdiff6 identifies the combination step, which merges different alignment techniques, as the key factor enabling improved patch compactness. Finally, we discuss the integration of bsdiff6 with OSTree and Consultative Committee for Space Data Systems (CCSDS) communication protocols to enable secure, verifiable, and bandwidth-efficient OTA updates for future space missions. Additionally, we provide an outlook on how our findings can advance research in the field of delta coding.**

I. Introduction

The recent dramatic increase in the number of satellites [1], driven in large part by the affordability of small platforms such as CubeSat [2], has diversified and broadened the range of organizations operating space assets and accelerated demand for sustainable in-orbit software maintenance. As missions become more numerous and diverse, the ability to update onboard software after launch is increasingly essential for maintaining security and reliability. Vulnerabilities discovered post-deployment, new payload features, or improved fault-handling routines can only be integrated if an effective OTA update mechanism exists. However, while the concept of OTA updates is well established for terrestrial devices like IoT devices [3], [4] and cars [5]–[7], its realization in the space domain faces distinct challenges. In the case of satellites, both the uplink bandwidth and the connection time windows are limited. These constraints emphasize the necessity for highly efficient delta-based update mechanisms that minimize the data volume required to keep a satellite secure and functional throughout its lifetime.

The trend of using COTS components, which has been long established for satellite hardware [8], is now expanding to software. This trend has prompted the adoption of satellite-tailored modified operating systems, which provide integrated services for telecommand, telemetry, and payload integration. Linux-based systems, in particular, are now seeing significant adoption across the space industry [9]–[13]. For instance, large constellations, such as Starlink [14], rely extensively on Linux for their operational software stack. Recent efforts toward standardized flight software environments, such as RCCN OS [15] (formerly RACCON OS), further reinforce this trend.

RCCN OS, unlike the core Flight System (cFS) [9] project, which is designed to run on top of any Linux distribution, provides a complete Linux distribution tailored specifically to spacecrafts and includes many of the components required for flexible satellite operations. Despite this solid foundation, it currently lacks a dedicated OTA update mechanism. A natural candidate for addressing this limitation, as proposed by its developers [16], is OSTree, a framework widely used in the embedded domain [17]–[19] for managing system snapshots, enabling atomic upgrades, and supporting rollback functionality.

OSTree internally relies on bsdiff version 4 (bsdiff4) [20] for binary delta generation. Although bsdiff4 has seen broad adoption [20]–[23], more recent studies [24]–[27] identify HDiffPatch [28] as the current state-of-the-art in diffing and delta encoding. This discrepancy raises a key question for space system design: *Given the unique bandwidth and communication constraints of satellites, should RCCN OS retain bsdiff4 through OSTree, or could a switch to more modern algorithms such as HDiffPatch, or even one of the bsdiff4 successors like bsdiff6 [29], of which an implementation was never published, and bsdiff-ra [30] significantly reduce the size of update payloads and help make satellites more secure?*

Workshop on Security of Space and Satellite Systems (SpaceSec) 2026
23 February 2026, San Diego, CA, USA
ISBN 978-1-970672-02-2
https://dx.doi.org/10.14722/spacesec.2026.23070
www.ndss-symposium.org

The contributions of this work are as follows:

1) We present the first publicly available *implementation of bsdiff6*.
2) We perform a comparative analysis of bsdiff4, bsdiff6, bsdiff-ra, and HDiffPatch.
3) We analyze the improvements introduced in bsdiff6 and discuss their potential to advance the state-of-the-art in delta compression.
4) We provide recommendations for OTA update mechanisms in space systems, using RCCN OS as a case study.

With this work, we aim to contribute to the design of future OTA update architectures for satellites, thereby enabling more bandwidth-efficient and secure software maintenance throughout the entire life cycle of a mission. Our implementation is publicly available at https://github.com/Julian-Rederle/bsdiff6.

II. Background and Related Work

In this section, we introduce the principles of binary diffing, also known as delta encoding or delta compression. We then outline the structure and mechanisms of representative algorithms, including bsdiff4, bsdiff6, bsdiff-ra, and HDiffPatch. Finally, we identify the research gaps this work addresses, particularly concerning the applicability of these algorithms to Over-the-Air (OTA) updates for space systems.

A. Binary Diffing and Delta Encoding

Delta encoding tools compute patches that transform one file version into another while minimizing the number of transmitted bytes. Most algorithms share a common three-stage structure:

1) **Alignment generation**, identifying matching regions between the old and new file.
2) **Patch construction**, encoding differences using copy instructions (derived from alignments) and add instructions (for unmatched regions).
3) **Compression**, reducing the final patch size, either through compression of the whole patch file or by compressing patch file sections individually.

An alignment indicates a region present in both files. This region may be identical or may contain small modifications.

B. Delta Encoding Algorithms

Below, we summarize the delta encoding algorithms examined in this work:

a) ***bsdiff4****:* bsdiff4 [31] aims to produce minimal patch sizes across diverse file formats by constructing a suffix array of the old file using `qsufsort` [32], enabling efficient longest-suffix matching against the new file. This local alignment approach identifies exact matches, which are subsequently extended to approximate matches in a tweaking step. The algorithm defines a custom patch format consisting of control, diff, and extra blocks, each individually compressed using `bzip2` [33]. The control block encodes copy instructions, while differences and unmatched regions are stored in the diff and extra blocks, respectively.

b) ***bsdiff6****:* bsdiff6 [29], the successor to bsdiff4, enhances the original algorithm by adding a block alignment step. This new step, based on a novel matching with mismatches approach, is performed in addition to the existing local alignment step. The core idea is that the local alignment approach excels at aligning small regions that match perfectly, while the block alignment approach can align larger regions with slight mismatches, thereby compensating for each other's weaknesses. With this approach yielding two sets of alignments, a combination step is introduced, merging the best alignments from both approaches into the final set of alignments used to create the patch file. Percival's notes suggest that this approach reduces average patch sizes by an average of 20% compared to bsdiff4 [34], though a full public implementation was never released.

c) ***bsdiff-ra****:* bsdiff-ra (random-access variant) [30] incorporates a block alignment strategy inspired by bsdiff6 but organizes the algorithms differently. Instead of combining two alignment sets, bsdiff-ra first applies block alignment to identify coarse-grained matches and then refines these results using local alignment.

d) ***HDiffPatch****:* HDiffPatch [28] is a modern, open-source delta encoding framework inspired by bsdiff4, introducing several engineering and algorithmic enhancements [24]. It replaces `qsufsort` with `DivSufSort` [35], achieving superior time and memory efficiency in suffix array construction [36]. Differences in the diff block are further encoded using Run Length Encoding (RLE), improving compression ratios. The tool allows variable dissimilarity thresholds for approximate matches depending on segment length and introduces a novel copy-instruction merging mechanism. Adjacent copy operations with small mismatches are merged into a single instruction, and the differing bytes are encoded via RLE in a dedicated control block. This approach reduces instruction overhead and improves compression friendliness.

C. Research Gap

Although delta encoding is a well-studied research area, we identify three specific gaps.

First, although the conceptual design of bsdiff6 is described by Percival in his thesis, no complete, reproducible implementation exists, preventing empirical evaluation and analysis of the design choices underlying its reported improvements.

Second, there is no published analysis of bsdiff-ra. Given that bsdiff4 is widely used, bsdiff6 claims substantial improvements, and bsdiff-ra reportedly[1] incorporates and extends aspects of bsdiff6, one would expect a rich body of comparative research.

Third, existing work on OTA updates for satellites focuses on secure transport and update coordination rather than the diffing layer itself. Prior studies [37]–[40] assume the availability of a delta encoding mechanism without evaluating its suitability under satellite-specific constraints such as limited uplink bandwidth, short contact windows, and mixed types of

[1] Based on https://github.com/cperciva/bsdiff/issues/2.

data found in satellite software stacks. Recent results showing HDiffPatch outperforming bsdiff4 [24], [27], [28], together with the continued reliance on bsdiff4 in systems such as OSTree, indicate that these assumptions are insufficient and motivate a systematic comparison.

Rather than proposing a domain-specific diffing algorithm, as pursued in other domains [21], [22], this work evaluates existing algorithms on space-relevant workloads. To address these gaps, we (i) construct a representative space-domain-specific input set (Section III-C), (ii) provide guidance for selecting suitable delta encoding algorithms (Section IV-B), and (iii) outline a complete OTA update design for spacecrafts (Section VI-A). Beyond this, we extend the state of the art by comparing HDiffPatch (the leading modern tool [24]–[27]) not only with bsdiff4 (the de facto baseline [20]–[23]) but also with bsdiff-ra and our reimplementation of bsdiff6. We analyse weaknesses of bsdiff-ra (Section IV-A), dissect the design choices underlying bsdiff6's performance advantages (Section V-B), and discuss implications for future research in the delta encoding domain (Section V-C).

III. Methodology

This section describes the experimental methodology, including the comparative analysis setup, tool configurations, and input dataset.

A. Comparative Analysis Methodology

The central metric of our evaluation is the **patch size**, representing the total number of bytes required to transmit. This metric directly corresponds to the transmission cost of an update. To conduct this comparison, we developed a Python-based analysis tool, automating the following tasks:

1) Iterate over all pairs of input file versions as selected in Section III-C.
2) Generating patch files for the input pair using each diffing algorithm configured as described in Section III-B.
3) Calculate the **space reduction ratio** as follows:

$$\text{space reduction} = \left(1 - \frac{\text{patch size}}{\text{new version size}}\right) \cdot 100$$

4) Summarize the result by averaging the space reduction ratios across all inputs and calculating the total bytes required to send all resulting patches.

B. Diff Tool Details

The patch generation process consists of three stages: alignment discovery between old and new files, packaging of aligned and unaligned regions into a patch, and compression of the resulting patch. Because the compression choice directly affects patch size and is largely interchangeable across tools, we eliminate the compression-related bias by using bzip2[2] for all four tools.

We kept the compression step, as uncompressed patch sizes alone would be misleading. This is because algorithms such as bsdiff6 explicitly trade raw patch size for improved compressibility by favouring repetitive structures.

HDiffPatch requires no special preparation beyond following the build instructions, as it supports bzip2 out of the box. For bsdiff4, we chose the original implementation by Colin Percival [30] over the commonly used refactored variant by Matthew Endsley [41] (used by OSTree), in order to remain closest to the original. bsdiff-ra was adapted for evaluation by applying minor build fixes and replacing its packaging logic with that of bsdiff4, which is possible due to their shared alignment representation and ensures that patch size differences are independent of the packaging format.

For bsdiff6, only a theoretical description is publicly available [29]. We obtained access to Percival's original C prototype but were not permitted to publish it, and therefore reimplemented bsdiff6 based on the prototype and its formal description. The implementation closely mirrors the original behaviour and is written in Rust to provide memory safety guarantees, avoiding known vulnerability classes affecting bsdiff4, such as CVE-2014-9862 [42] and CVE-2020-14315 [43].

C. Dataset and Input Selection

The input dataset is designed to represent both operating system and application-level components commonly found in satellite software stacks, as well as data typically produced by payloads. Its composition is inspired by prior research [24], [44] on delta compression and guided by the distinctive characteristics of spaceborne and embedded systems.

Our dataset selection consists of the following:

*a) **Executables**:* This category comprises two versions each of the Linux Kernel, gcc (and g++), BusyBox, libcurl (as a Linux tool), glibc (commonly used), SQLite (data storage domain), OpenSSL (security domain), and OpenCV (image processing domain). These represent the characteristics of executable file formats, operating system and application components, as well as tools used for different operational aspects such as security-related or payload-related software.

*b) **Configuration Files**:* Two versions of a U-Boot configuration file and a Docker Compose YAML file configured to deploy a GitLab instance. These files illustrate the characteristics and challenges specific to text-based configuration formats.

*c) **Other Text-based Files**:* This category includes two versions of the file `polynomial.py` from the NumPy library, representing Python source code. Python, alongside Rust, is one of the commonly used programming languages within the RCCN OS software stack.

*d) **Other File Types**:* As Linux setups frequently utilize disk images, two versions of a Raspberry Pi OS image are selected. These are included in two common deployment formats: an `xz` archive and an unpacked `img` file. Additionally, we include both a PNG and a JPEG image, each with minor pixel adjustments. While this does not represent a routine update scenario, such files are relevant as examples of potential

[2]Chosen as it is the standard for the bsdiff tools and a supported option in HDiffPatch.

payload data (e.g.,, from a camera) and are noteworthy because these formats already include a degree of internal redundancy similar to that of compressed archives.

*e) **APK**:* Sun et al. [24], one of the most recent studies on delta compression performance, as well as HDiffPatch [28], both employ input datasets focused on the Android Package Kit (APK) format. APK files are compelling as input data because they are essentially ZIP archives containing multiple files and directories, meaning that a significant portion of internal redundancy has already been reduced. Following this line of research, we included two versions each of the WhatsApp, Firefox, and Instagram APKs[3], selected from the most downloaded applications on Google Play. Additionally, we incorporated the x86 Linux installer for Discord to include `gzip` as an additional compression format.

IV. Results

In this section, we present the results of our analysis and put them into perspective.

A. Quantitative Comparison

Table I summarizes the patch size results for all evaluated algorithms across the selected binaries. In total, nineteen input pairs were analyzed, covering a diverse range of software artifacts. Across all inputs, bsdiff6 produced the smallest patches in 11 out of 19 cases. To contextualize the results, we consider bsdiff4, the widely used but now dated algorithm, as the baseline and reference. All relative reductions and improvements are expressed with respect to their patch sizes, as shown in Table II. This framing enables the comparison of alternative algorithms from a consistent perspective.

The only input that does not allow for comparison is the `Raspberry Pi img`. All bsdiff versions utilize signed 32-bit integers whose value range does not support addressing the ≈2,6GB of the `Raspberry Pi img`. Thus, the only tool capable of creating a patch for this input is HDiffPatch. The resulting patch is ≈110MB in size, which is a reduction of ≈95,68%.

For the other inputs, we observe the following:

*a) **Released bsdiff4 (Reference)**:* As seen in Table I, size reductions vary strongly depending on the input data structure, from over 80% for highly structured data[4] (e.g., glibc, curl, and OpenCV) down to below 10% for formats that already remove a certain level of redundancy (e.g., Discord gzip, Cat.png, and Research.jpg). These results confirm the known behavior of bsdiff4 and approaches based on it: its suffix matching approach achieves excellent compression for executables and other structured data, but yields limited benefit when redundancy is low or content is already compressed.

*b) **Released bsdiff-ra (Random Access Variant)**:* In our analysis, bsdiff-ra performs nearly identically to bsdiff4 on most inputs, with an average increase in the resulting patch size of +0.9%. However, in some cases the sizes increase dramatically, in the worst case by more than 100% (e.g., OpenCV). In addition, bsdiff-ra crashes for the inputs `gcc` and `WhatsApp APK`, preventing us from calculating a comparable total number of bytes for all patches, as well as skewing the average reduction value due to the missing data point.

These findings are somewhat expected due to the algorithm's design limitations. The local alignment step is restricted to aligning within the coarse-grained aligned blocks, without a mechanism for merging or further tweaking the alignments of neighboring blocks. Based on the analysis results, the algorithm performs similarly to bsdiff4 in cases where the blocks are aligned correctly in the first step, and worse when blocks are not aligned correctly and the fine-grained local alignment step can only search for alignments in an incorrect region.

These findings show that bsdiff-ra does not provide any improvement over bsdiff4, neither in patch compactness nor robustness.

*c) **Our bsdiff6 (The bsdiff4 Successor)**:* Our re-implementation of bsdiff6 in Rust demonstrates consistent improvements in patch compactness relative to bsdiff4. Across all inputs, bsdiff6 reduces patch size by an average of ≈ 4,8%, outperforming bsdiff4 in 18 of 19 cases. The only exception, where bsdiff6 produces a patch larger than the one bsdiff4 produces is `polynomial.py`, with a negligible increase of 59 bytes.

These reductions align with the design intent of bsdiff6, improving bsdiff4 by addressing inefficiencies in the local alignment approach.

*d) **HDiffPatch (State-of-the-Art)**:* HDiffPatch, compared to bsdiff4, achieves an average patch size reduction of 6,6%, exceeding bsdiff6's gain. In terms of patch size reduction, it only outperforms bsdiff4 for 15 of 19 inputs. For `Buysbox`, the `Linux Kernel`, `curl`, and `Instagram APK` inputs, it performs worse than bsdiff4.

Especially noteworthy is the size reduction performance for the `Raspberry Pi xz` versions, where HDiffPatch reduces the patch size by ≈2MB compared to bsdiff4 and ≈1,6MB compared to bsdiff6. In general, there is a trend towards HDiffPatch outperforming the other algorithms for file types that have a certain level of internal redundancy reduction in place (e.g., gzip, xz, JPG, and PNG). Interestingly, this pattern of outperforming the other algorithms does not hold for APKs, despite APKs being the main focus of investigations conducted by the developers of HDiffPatch [28] and others [24].

B. Qualitative Comparison of Leading Candidates

Given their superior overall performance, bsdiff6 and HDiffPatch represent the most promising approaches among the evaluated algorithms, but their strengths differ by use case. bsdiff6 consistently outperforms bsdiff4 for nearly all inputs, making it a clear successor and a practical drop-in replacement

[3]The APK versions are obtained from the archive at `www.apkmirror.com`.

[4]For instance, ELF files contain a header structured that is consistent across file versions.

Input (from → to)	Old Size in bytes	New Size in bytes	Released bsdiff4 Patch Size in bytes	Released bsdiff4 Reduction in %	Released bsdiff-ra Patch Size in bytes	Released bsdiff-ra Reduction in %	Our bsdiff6 Patch Size in bytes	Our bsdiff6 Reduction in %	HDiffPatch Patch Size in bytes	HDiffPatch Reduction in %
Busybox (1.35.0 → 1.36.0)	1.084.088	1.063.528	295.084	72.25%	295.084	72.25%	292.213	72.52%	303.519	71.46%
glibc (2.38 → 2.39)	2.676.552	2.307.728	258.100	88.82%	257.977	88.82%	201.661	91.26%	221.508	90.40%
Linux Kernel (6.6 → 6.7)	12.814.400	12.874.752	12.688.134	1.45%	12.752.962	0.95%	12.680.438	1.51%	12.689.790	1.44%
OpenSSL (3.0.4 → 3.4.0)	1.143.744	1.255.640	301.440	75.99%	301.440	75.99%	261.418	79.18%	282.964	77.46%
SQLite (3.44.2 → 3.47.0)	7.170.048	7.485.272	1.749.836	76.62%	2.034.584	72.82%	1.585.251	78.82%	1.660.404	77.82%
curl (8.10.1 → 8.11.0)	473.672	473.672	74.940	84.18%	74.940	84.18%	74.810	84.21%	75.467	84.07%
gcc (13.3.0 → 14.2.0)	7.671.392	9.151.784	2.699.108	70.51%	Tool Error	–	2.391.807	73.87%	2.541.066	72.23%
g++ (13.3.0 → 14.2.0)	7.682.536	9.162.712	2.703.309	70.50%	2.938.527	67.93%	2.399.561	73.81%	2.546.240	72.21%
OpenCV (4.8.1 → 4.10.0)	15.579.056	15.785.560	1.689.891	89.29%	3.400.030	78.46%	1.489.539	90.56%	1.550.220	90.18%
Whatsapp APK (2.24.24.79 → 2.24.25.72)	225.781	105.079.745	96.653.979	8.02%	Tool Error	–	96.577.634	8.09%	96.538.922	8.13%
Firefox APK (132.0 → 133.0)	105.433.914	106.471.156	80.817.691	24.09%	82.841.332	22.19%	80.339.977	24.54%	80.564.334	24.33%
Instagram APK (403.0.0.49.74 → 404.0.0.48.76)	5.929.780	5.943.329	564.208	90.51%	572.728	90.36%	536.577	90.97%	565.486	90.49%
Discord gzip (0.0.58 → 0.0.76)	103.212.154	102.127.793	101.634.220	0.48%	102.348.801	-0.22%	101.561.238	0.55%	101.364.634	0.75%
Raspberry Pi xz (2024-11-13 → 2024-11-19)	535.114.896	532.543.404	432.758.855	18.74%	455.827.018	14.41%	432.412.353	18.80%	430.778.964	19.11%
Numpy polynomial.py (2.0.0 → 2.1.3)	52.572	52.699	535	98.98%	535	98.98%	594	98.87%	424	99.20%
Gitlab Docker compose (8.0.4 → 8.9.4)	1.414	3.694	1.132	69.36%	1.132	69.36%	1.103	70.14%	949	74.31%
u-boot config (x86 App32 → x86 App64)	1.041	1.187	297	74.98%	297	74.98%	286	75.91%	160	86.52%
Cat.png (default → red box)	133.650	119.757	118.248	1.26%	118.248	1.26%	118.237	1.27%	118.142	1.35%
Research.jpg (Red → Blue)	211.446	292.043	277.369	5.02%	277.369	5.02%	277.358	5.03%	277.237	5.07%
Total bytes		912.195.455	735.286.376		–		733.202.055		732.080.430	
Average reduction				53.74%		53.99%		54.73%		55.08%

TABLE I
COMPARISON BETWEEN THE ORIGINAL BSDIFF4 IMPLEMENTATION, THE SUCCESSOR BSDIFF-RA, OUR IMPLEMENTATION OF BSDIFF6 AND THE STATE-OF-THE-ART ALGORITHM HDIFFPATCH.

in systems such as `OSTree`. This substitution would directly reduce the amount of data transmitted during updates without requiring architectural changes.

Design considerations, however, strongly influence the preferred choice. HDiffPatch performs worse on typical system binaries, such as `Busybox` and the `Linux Kernel`, but excels for large, compressed images, like the `Raspberry Pi xz`. Thus, for a modular system where components are updated individually, bsdiff6 is the better fit. For monolithic designs that distribute entire OS images, HDiffPatch provides better performance and is the only tool that can create a patch for the uncompressed `Raspberry Pi img`, also supports larger input files, as is typical when dealing with complete OS images.

Finally, neither total bytes nor average reduction values are reliable indicators, as single outliers (such as the `xz` image) can substantially skew the results. Thus, the choice between algorithms should be guided by the update granularity of the satellite and the characteristics of the data, rather than overall averages.

V. DESIGN IMPLICATIONS

The results of our comparative analysis demonstrate that bsdiff6 achieves a clear performance advantage in terms of patch size reduction compared to bsdiff4. With this knowledge in hand, we are interested in identifying the underlying factors that enable `bsdiff6` to outperform its predecessor. The PhD thesis [29] introducing the concepts of bsdiff6 attributes this improvement to the introduction of a matching-with-mismatches approach, realized through an additional block alignment component.

During the development and testing of our bsdiff6 reimplementation, however, we observed that errors in the block alignment step rarely led to a measurable decrease in patch size performance. Instead, our results indicate that the combination step is the key factor responsible for bsdiff6's superior performance. In the following, we discuss this finding in greater depth.

A. High-level Overview

Both versions of the `bsdiff` family are composed of several largely independent stages, allowing for a decomposition

Input	Released bsdiff4 Size reference in bytes	Released bsdiff-ra Reduction in bytes	Released bsdiff-ra Reduction in %	Our bsdiff6 Reduction in bytes	Our bsdiff6 Reduction in %	HDiffPatch Reduction in bytes	HDiffPatch Reduction in %
Busybox	295.084	0	0,0%	-2.871	-0,97%	+8.435	+2,86%
glibc	258.100	-123	-0,05%	-56.439	-21,87%	-36.592	-14,18%
Linux Kernel	12.688.134	+64.828	+0,51%	-7.696	-0,06%	+1.656	+0,01%
OpenSSL	301.440	0	0,0%	-40.022	-13,28%	-18.476	-6,13%
SQLite	1.749.836	+284.748	+16,27%	-164.585	-9,41%	-89.432	-5,11%
curl	74.940	0	0,0%	-130	-0,17%	+527	+0,7%
gcc	2.699.108	Error	Error	-307.301	-11,39%	-158.042	-5,86%
g++	2.703.309	+235.218	+8,7%	-303.748	-11,24%	-157.069	-5,81%
OpenCV	1.689.891	+1.710.139	+101,2%	-200.352	-11,86%	-139.671	-8,27%
Whatsapp APK	96.653.979	Error	Error	-76.345	-0,08%	-115.057	-0,12%
Firefox APK	80.817.691	+2.023.641	+2,5%	-477.714	-0,59%	-253.357	-0,31%
Instagram APK	564.208	+8.520	+1,51%	-27.631	-4,9%	+1.278	+0,23%
Discord gzip	101.634.220	+714.581	+0,7%	-72.982	-0,07%	-269.586	-0,27%
Raspberry Pi xz	432.758.855	+23.068.163	+5,33%	-346.502	-0,08%	-1.979.891	-0,46%
Numpy polynomial.py	535	0	0,0%	+59	+11,03%	-111	-20,75%
Gitlab Docker compose	1.132	0	0,0%	-29	-2,56%	-183	-16,17%
u-boot config	297	0	0,0%	-11	-3,7%	-137	-46,13%
Cat.png	118.248	0	0,0%	-11	-0,01%	-106	-0,09%
Research.jpg	277.369	0	0,0%	-11	-0,0%	-132	-0,05%
Total byte reduction		-107		-2.084.379		-3.217.838	
Average reduction			+0,94%		-4,8%		-6,62%

TABLE II
OVERVIEW OF INCREASE (OR DECREASE) IN PATCH SIZE PERFORMANCE OF BSDIFF-RA, BSDIFF6 AND HDIFFPATCH COMPARED TO BSDIFF4 BASED ON RESULTS PRESENTED IN TABLE I.

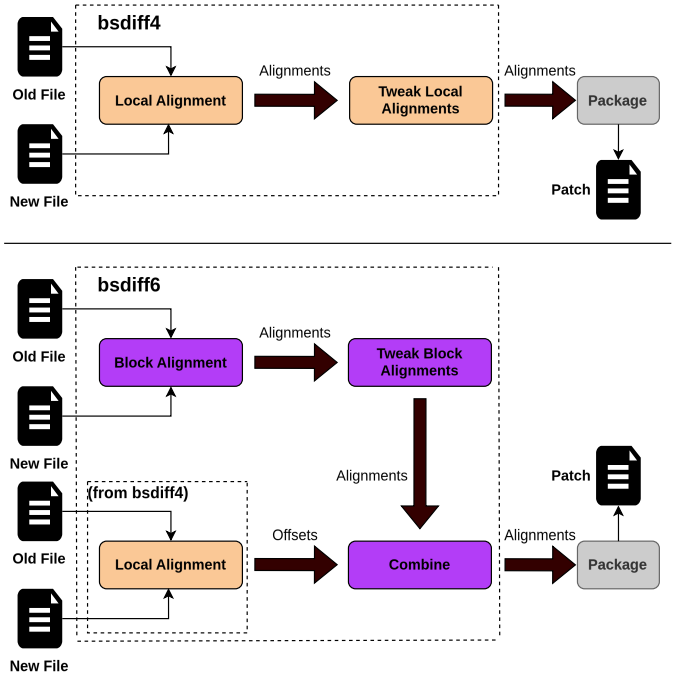

Fig. 1. Overview and side-by-side comparison of the bsdiff4 and bsdiff6 building blocks.

of the algorithms into modular components, as illustrated in Figure 1. We do not consider the patch packaging step as part of the diffing algorithm itself, as it remains identical across implementations and could, in principle, be replaced by any alternative mechanism that consumes the same intermediate alignment representation.

bsdiff4 consists of a local alignment step that produces a set of alignments, which are subsequently refined during a tweaking step before being packaged into a patch file.

bsdiff6, by contrast, introduces a block alignment step that implements the matching-with-mismatches approach. The resulting alignments are tweaked by moving their block boundaries. This is done, as the underlying structure of files can rarely be described by equally sized blocks. A local alignment approach is also applied to the file versions. While bsdiff4's local alignment algorithm identifies the longest matching suffix for an index of the new version and skips indices in already aligned regions, bsdiff6's variant does not skip indices. Instead, it searches for each byte of the new file, the offset in the old file where the best match occurs, using the same suffix approach. The outputs of the block alignment and local alignment steps are subsequently combined to leverage the strengths of both approaches, forming the final alignments consumed by the patch packaging step.

B. Combination Step

In theory, an optimal diff could be produced by aligning every byte of the new file against every possible position in the old file and selecting the combination of alignments

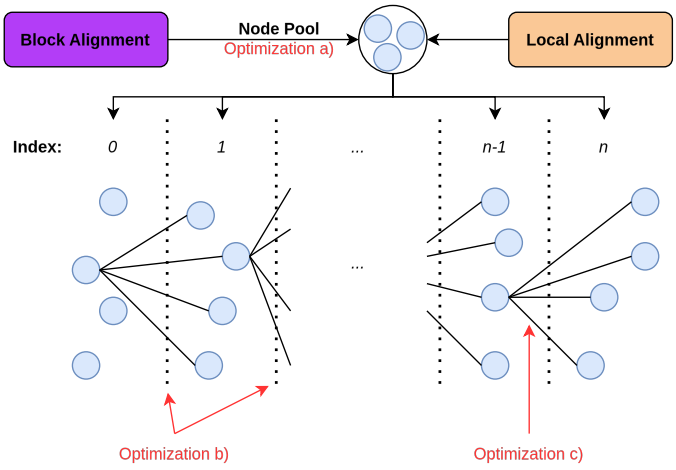

Fig. 2. Visualization of the improvements making the combination step computationally feasible.

that minimizes patch size. This problem can be formulated as a graph search, where each node represents a potential alignment with a score reflecting its match quality. Nodes representing adjacent indices in the new version are connected by edges, yielding a weighted graph through which algorithms such as Dijkstra's shortest path [45] could identify the optimal sequence of alignments. However, such an approach is computationally infeasible, as the number of nodes, edges, and possible paths grows exponentially with input size.

Our analysis shows that the bsdiff6 combination step effectively approximates this optimal strategy while maintaining computational feasibility through three key optimizations, visualized in Figure 2:

a) ***Reduction of Node Count****:* Instead of constructing a graph with all possible alignments, bsdiff6 restricts the node set to the outputs of the block and local alignment steps, representing a pool of likely optimal alignments. This dramatically reduces computational effort, albeit at the potential cost of excluding the optimal solution. Nevertheless, our experiment results indicate that this approximation is sufficient to significantly reduce patch sizes compared to the prior bsdiff4 approach.

b) ***Layer-by-Layer Graph Construction****:* Since edges exist only between nodes of adjacent indices, the graph can be constructed incrementally, one layer at a time, for each index of the new file. Each layer connects only to the previous one, eliminating the need to keep the full graph in memory. This layered approach is especially interesting for the next optimization.

c) ***Selective Edge Connections****:* With the layer-by-layer design, bsdiff6 can save on edges by only connecting each node of the current layer to the best-scoring node of the previous layer. This effectively prunes the search space, leaving a single valid path from the last to the first index of the new file. As a result, the need for an explicit shortest path computation is removed entirely, reducing computational cost to linear complexity with respect to file size.

In addition to these optimizations, the combination step introduces mechanisms such as score penalties and the option to allow regions not to be aligned, which further improve patch size reduction. These details, including the description of the penalty mechanism, fall beyond the scope of this paper but are documented alongside the released code.

C. Implications for State-of-the-Art

Since our results suggest that the combination step is the improving factor, the question remains as to what this means for the current state-of-the-art. Analysis of the size reduction performance differences and implementation details suggests that the combination step alone can dramatically improve alignment quality, thereby reducing the resulting patch sizes by refining the alignments.

While the current bsdiff6 implementation limits its refinement process to alignments derived from block- and local-alignment approaches, future work could generalize this component to integrate candidate alignments from other approaches such as HDiffPatch. A generalized, modular combination framework could therefore serve as a uniform refinement stage across many diffing tools.

Finally, the current penalty system in bsdiff6 is tuned for bzip2 compression. To reach optimal performance with modern compression tools such as LZMA, future studies should investigate the relationship between penalties and compression efficiency. Such analyses could yield adaptive penalty models that dynamically optimize patch generation based on the characteristics of the inputs and chosen compression.

VI. Discussion

A. Complete Design Suggestion

Although not the primary focus of this work, we outline a secure, verifiable, and bandwidth-efficient OTA update design for space systems, using RCCN OS as a case study. In line with recent design choices [16] and related approaches [37]–[40], we propose combining OSTree for versioned system management, bsdiff6 for binary delta compression, and standardized CCSDS protocols for secure data transport. The security guarantees of the design are provided by the CCSDS protocol suite. Analysis of these protocols is therefore considered outside the scope of this work.

OSTree offers a git-like, content-addressed storage model with atomic upgrades and rollback functionality by representing system states as signed commits, and provides built-in authenticity and integrity verification via GPG keys [46].

Patch transmission from ground to spacecraft targets confidentiality, authentication, and disruption tolerance. These properties are achieved using CCSDS File Delivery Protocol (CFDP) [47] via Licklider Transmission Protocol (LTP) [48], which supports resumable partial transfers through checkpointing [49], combined with Space Data Link Security (SDLS) for link-layer encryption and authentication [50].

For delta generation, we recommend replacing the bsdiff4 backend used by OSTree with bsdiff6. As a drop-in replacement compatible with the bsdiff4 patch format, bsdiff6

requires no changes to patch application logic while reducing exposure to memory-safety vulnerabilities through its Rust-based implementation. In addition, bsdiff6 outperforms bsdiff4 in terms of patch size for all but one evaluated input, resulting in a definite increase in efficiency.

Within OSTree's commit-based update model, diffing primarily targets system binaries rather than complete OS images, a scenario where bsdiff6 shows superior performance (Section IV-B). Nonetheless, further evaluation of alternative compression algorithms and penalty parameter tuning in the combination step is recommended before deployment.

Finally, RCCN OS has integrated OSTree-based OTA support for some time. At the time of writing, and although not yet officially documented, RCCN OS has begun developing CFDP and SDLS support for their latest mission, CyBEEsat [51].

B. Patch Size Reduction Claims

A key point of discussion concerns our observation that bsdiff6 produces, on average, patches approximately 4,8% smaller than those generated by bsdiff4. This result does not align with Colin Percival's original claim of a $\approx 20\%$ improvement [34], which was further supported by Motta et al. [44]. While patch size reduction percentages are input dependent, our experiments yielded only one case approaching 20%, four around 10%, and the remainder below 5%. Consequently, we cannot confirm the previously stated $\approx 20\%$ reduction.

Although the exact reason for this discrepancy remains unclear, our analysis suggests that differences in patch packaging implementations may be a contributing factor. The prototype of bsdiff6 available to us lacked certain packaging features, indicating that the packaging logic may have evolved over time. We therefore assume that the bsdiff4 implementation used in our comparison differs from the version originally used for the 20% claim, and that this claim may not have been based on a consistent comparison using identical packaging mechanisms.

C. Limitations of the Analysis Methods

Patch size reduction, total byte count, and compression ratio are common performance metrics for evaluating delta compression algorithms [24], [29], [31], [44]. However, our evaluation indicates that these metrics can be misleading. The choice of input pairs has a significant influence on the results, sometimes leading to incorrect conclusions.

For example, Table I suggests that `HDiffPatch` performs substantially better, with a reduction of ≈1,1MB. In reality, this advantage is primarily due to the `xz` input pair, where the HDiffPatch output is ≈1,6MB smaller than the bsdiff6 patch. When this specific input is excluded, bsdiff6 outperforms HDiffPatch for this metric.

Even our comparison approach, which uses bsdiff4 as a baseline (Table II), suffers from similar limitations. For instance, the bsdiff6 result for `polynomial.py` increases patch size by only 59 bytes, yet skews the average percentage increase by ≈11%. In contrast, HDiffPatch achieves a 20,75% reduction for the same input, the second highest observed improvement, illustrating how small variations can distort the metrics.

This issue appears to have received limited attention in prior literature, suggesting that future research should focus on developing a standardized and universally applicable framework for performance evaluation in binary diffing.

Beyond metric-related concerns, existing research typically evaluates complete tools rather than isolating individual components. While this type of assessment is suitable for determining the best-performing tool for a given use case, studies aiming to improve specific algorithmic aspects, such as the work by Sun et al. [24], should evaluate diffing logic, patch packaging, and compression mechanisms independently. Without such separation, performance gains due to differences in diffing algorithms cannot be reliably distinguished from those arising from differences in compression performance.

While our methodology decouples compression effects, we could not fully isolate the impact of patch packaging, as only the bsdiff variants share a common format, whereas HDiffPatch uses its own. Thus, establishing a standardized intermediate alignment representation would enable interoperability across diffing, packaging, and compression schemes. This would, for example, allow bsdiff6 to leverage HDiffPatch's RLE and copy-merge optimizations to further reduce patch size.

D. Statistical Considerations

Our input set selection process was inspired by related work [24], [28], [31], [44], but with a more specialised focus on the space domain. While this related work reflects the general acceptance of evaluations based on relatively small input sets, it is common in other research areas to rely on larger datasets to obtain more statistically meaningful results. Increasing the size of the input set would enable the application of statistical methods, such as confidence interval estimation. These methods would enable stronger and more general statements about performance improvements beyond the observed differences resulting from our inputs.

E. Future Work

Future work should evaluate the bsdiff6 combination step, particularly its penalty parameters and their interaction with alternative compression algorithms such as LZMA, to better understand the separation between diffing and compression. Further analysis of interactions between diff generation stages could reveal whether components are interchangeable across approaches, for example by reusing HDiffPatch alignments as input to the bsdiff6 combination step. Finally, the scalability of bsdiff6 for multi-gigabyte inputs and its integration into operational OTA frameworks such as RCCN OS should be validated in real-world deployments.

VII. Conclusion

This work presented the first publicly available reimplementation of bsdiff6, reconstructed from Colin Percival's original design notes and prototype. Through a comparative analysis,

we demonstrated that bsdiff6 consistently reduces patch sizes relative to bsdiff4 while maintaining compatibility and security advantages through its Rust-based implementation. Our findings identify the combination step as the key innovation behind bsdiff6's improvements and highlight its potential to advance the state of delta encoding. Finally, by integrating bsdiff6 with OSTree and CCSDS communication protocols, we outlined a practical design for secure, verifiable, and bandwidth-efficient OTA updates in satellite systems.

Acknowledgment

The authors would like to thank Colin Percival, the original creator of bsdiff4 and bsdiff6, for his foundational work in binary diffing and for kindly providing access to his prototype implementation of bsdiff6, which served as an invaluable reference for this research. The project on which this report is based was supported with funding from the Federal Ministry of Research, Technology and Space (BMFTR) under the grant number 50YB2607B. The author is responsible for the content of this publication.

References

[1] United Nations Office for Outer Space Affairs (UNOOSA), "Policy brief 7: for all humanity – the future of outer space governance," UNOOSA. [Online]. Available: https://www.unoosa.org/res/oosadoc/data/documents/2023/a77/a77crp_1add_6_0_html/A_77_CRP01_Add06E.pdf

[2] The CubeSat Program, California Polytechnic State University, "Cubesat design specification," accessed: 2025-11-11. [Online]. Available: https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/62193b7fc9e72e0053f00910/1645820809779/CDS+REV14_1+2022-02-09.pdf

[3] C.-Y. Park, S.-J. Lee, and I.-G. Lee, "Secure and lightweight firmware over-the-air update mechanism for internet of things," *Electronics*, vol. 14, no. 8, p. 1583, 2025.

[4] F. Mahfoudhi, A. K. Sultania, and J. Famaey, "Over-the-air firmware updates for constrained nb-iot devices," *Sensors*, vol. 22, no. 19, p. 7572, 2022.

[5] S. Halder, A. Ghosal, and M. Conti, "Secure over-the-air software updates in connected vehicles: A survey," *Computer Networks*, vol. 178, p. 107343, 2020.

[6] H. Guissouma, C. P. Hohl, F. Lesniak, M. Schindewolf, J. Becker, and E. Sax, "Lifecycle management of automotive safety-critical over the air updates: A systems approach," *IEEE Access*, vol. 10, pp. 57 696–57 717, 2022.

[7] A. Shoker, F. Alves, and P. Esteves-Verissimo, "Scaiota: Scalable secure over-the-air software updates for vehicles," in *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2023, pp. 151–161.

[8] N. Yadav, F. Vollmer, A.-R. Sadeghi, G. Smaragdakis, and A. Voulimeneas, "Orbital shield: Rethinking satellite security in the commercial off-the-shelf era," in *2024 Security for Space Systems (3S)*. IEEE, 2024, pp. 1–11.

[9] D. McComas, J. Wilmot, and A. Cudmore, "The core flight system (cfs) community: Providing low cost solutions for small spacecraft," in *Annual AIAA/USU conference on small satellites*, no. GSFC-E-DAA-TN33786, 2016.

[10] H. Leppinen, "Current use of linux in spacecraft flight software," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 10, pp. 4–13, 2017.

[11] J. Couluris and T. Garvey, "Spacex mission operations," in *SpaceOps 2010 Conference Delivering on the Dream Hosted by NASA Marshall Space Flight Center and Organized by AIAA*, 2010, p. 1937.

[12] A. S. Ahmed Farid, A. Shalaby, A. Tarek, M. Ayyad, M. Assem, and S. Amin, "Utilizing low-cost linux micro-computer & android phone solutions on cube-satellites," 2013.

[13] C. Huffine, "Linux on a small satellite," *Linux Journal*, vol. 2005, no. 132, p. 9, 2005.

[14] G. Davis. (2022) Spacex starlink's weirdest facts: Open-source linux os and other peculiar things about the satellite. Accessed: 2025-11-14. [Online]. Available: https://www.techtimes.com/articles/285485/20221227/spacex-starlinks-weirdest-facts-open-source-linux-os-peculiar-things.htm

[15] R. O. Developers, "Raccoon os," https://gitlab.com/rccn, 2025, accessed: 2025-10-16.

[16] M. Ostasz, E. Stoll, P. Wüstenberg, J. Freymuth, J. M. D. López, and J. Izak, "Cybersecurity management of small satellites as a joint endeavour," in *Space Resources Conference*. Springer, 2024, pp. 119–128.

[17] M. Apetroaie-Cristea, "A modular and open software and hardware architecture for internet of things sensor networks," Ph.D. dissertation, University of Southampton, 2020.

[18] S. J. Johnston, P. J. Basford, F. M. Bulot, M. Apetroaie-Cristea, G. L. Fosterx, M. Loxhamz, and S. J. Cox, "Iot deployment for city scale air quality monitoring with low-power wide area networks," in *2018 Global Internet of Things Summit (GIoTS)*. IEEE, 2018, pp. 1–6.

[19] S. Stević, V. Lazić, M. Z. Bjelica, and N. Lukić, "Iot-based software update proposal for next generation automotive middleware stacks," in *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE, 2018, pp. 1–4.

[20] OSTree Developers, "Ostree." [Online]. Available: https://github.com/ostreedev/ostree

[21] The Chromium Project. Software updates: Courgette. Google Chromium Project. [Online]. Available: https://www.chromium.org/developers/design-documents/software-updates-courgette/

[22] Google Inc., "Archive patcher," Google. [Online]. Available: https://github.com/google/archive-patcher

[23] FreeBSD Project, "Portsnap: Provides secure snapshots of the ports directory," FreeBSD Project. [Online]. Available: https://www.freshports.org/sysutils/portsnap/

[24] T. Sun, B. Jiang, L. Jin, W. Zhang, Y. Gao, Z. Li, and W. Dong, "Understanding differencing algorithms for mobile application updates," *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 12 146–12 160, 2024.

[25] R. Lei, X. Chen, D. Liu, C. Song, Y. Tan, and A. Ren, "Optimizing the incremental update mechanism by inlaying file indexes on flash storage," in *2023 IEEE 12th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2023, pp. 56–61.

[26] L. Jin, W. Dong, B. Jiang, T. Sun, and Y. Gao, "Exploiting multiple similarity spaces for efficient and flexible incremental update of mobile apps," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 1541–1550.

[27] E. Moqvist, "Detools," Open Source. [Online]. Available: https://github.com/eerimoq/detools

[28] SiSong, "Hdiffpatch." [Online]. Available: https://github.com/sisong/HDiffPatch

[29] C. Percival, "Matching with mismatches and assorted applications," Ph.D. dissertation, University of Oxford, 2006.

[30] ——, "bsdiff." [Online]. Available: https://github.com/cperciva/bsdiff

[31] ——, "Naïve differences of executable code," 2003. [Online]. Available: http://www.daemonology.net/bsdiff/

[32] N. J. Larsson and K. Sadakane, "Faster suffix sorting," *Theoretical Computer Science*, vol. 387, no. 3, pp. 258–272, 2007.

[33] J. Seward, "bzip2 and libbzip2," *avaliable at http://www. bzip. org*, pp. 8–18, 1996.

[34] C. Percival, "Binary diff/patch utility," 2003, accessed: 2025-10-11. [Online]. Available: https://www.daemonology.net/bsdiff/

[35] Y. Mori, "libdivsufsort: A lightweight suffix-sorting library." [Online]. Available: https://github.com/y-256/libdivsufsort

[36] J. Fischer and F. Kurpicz, "Dismantling divsufsort," *arXiv preprint arXiv:1710.01896*, 2017.

[37] A. Badshah, N. Morris, and M. Monson, "Over-the-vacuum update–starlink's approach for reliably upgrading software on thousands of satellites," 2023.

[38] M. Cho and M. I. Monowar, "Over-the-air firmware update for an educational cubesat project," *International Review of Aerospace Engineering (IREASE)*, vol. 14, p. 39, 2021.

[39] A. Gangwal and A. Paliwal, "Csum: A novel mechanism for updating cubesat while preserving authenticity and integrity," in *2024 IEEE 49th Conference on Local Computer Networks (LCN)*. IEEE, 2024, pp. 1–7.
[40] D. Donsez, O. Alphand, F.-X. Molina, K. Zandberg, and E. Baccelli, "Cubedate: Securing software updates in orbit for low-power payloads hosted on cubesats," 2022.
[41] Mendsley, "bsdiff." [Online]. Available: https://github.com/mendsley/bsdiff
[42] "CVE-2014-9862: Integer signedness error in bspatch.c in bspatch in bsdiff, as used in apple os x before 10.11.6 and other products, allows remote attackers to execute arbitrary code or cause a denial of service (heap-based buffer overflow) via a crafted patch file." https://nvd.nist.gov/vuln/detail/CVE-2014-9862, 2014, accessed: 2025-10-22.
[43] "CVE-2020-14315: A memory corruption vulnerability is present in bspatch as shipped in colin percival's bsdiff tools version 4.3. insufficient checks when handling external inputs allows an attacker to bypass the sanity checks in place and write out of a dynamically allocated buffer boundaries." https://security-tracker.debian.org/tracker/CVE-2020-14315, 2020, accessed: 2025-10-22.
[44] G. Motta, J. Gustafson, and S. Chen, "Differential compression of executable code," in *2007 Data Compression Conference (DCC'07)*. IEEE, 2007, pp. 103–112.
[45] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: his life, work, and legacy*, 2022, pp. 287–290.
[46] OSTree Developers. libostree. OSTree Project. [Online]. Available: https://ostreedev.github.io/ostree/
[47] Consultative Committee for Space Data Systems (CCSDS), "Ccsds 727.0-b-5: Ccsds file delivery protocol (cfdp)." [Online]. Available: https://ccsds.org/Pubs/727x0b5e1.pdf
[48] ——, "Ccsds 720.2-g-4: Ccsds file delivery protocol (cfdp) part 2: Implementers guide." [Online]. Available: https://ccsds.org/Pubs/720x2g4.pdf
[49] ——, "Ccsds 734.1-b-1: Licklider transmission protocol (ltp) for ccsds." [Online]. Available: https://ccsds.org/Pubs/734x1b1.pdf
[50] ——, "Ccsds 355.0-b-2: Space data link security protocol." [Online]. Available: https://ccsds.org/Pubs/355x0b2.pdf
[51] RCCN Development Team, "Cybeesat mission repository," RCCN. [Online]. Available: https://gitlab.com/rccn/missions/cybeesat
[52] Space Wizards, "bsdiff-rs," Space Wizards. [Online]. Available: https://github.com/space-wizards/bsdiff-rs
[53] Divvun Group, "Bidiff," Divvun Group. [Online]. Available: https://github.com/divvun/bidiff

APPENDIX A
REIMPLEMENTATION DIFFERENCES

With this work focusing on a publishable reimplementation of bsdiff6 rather than the original prototype, potential reproducibility concerns must be addressed. A key challenge is balancing the goal of faithfully reproducing the algorithm with the need for maintainable and reliable code. The original C implementations of bsdiff4 and bsdiff6 make limited use of external libraries, instead providing custom implementations for core components such as the Greatest Common Divisor, Fast Fourier Transform, and functions related to the Suffix Array. In contrast, our approach prioritizes robustness by relying on tested libraries for these functions, which are typically more stable and less error-prone.

Because bsdiff6 builds upon parts of bsdiff4, our initial step was to establish an implementation of the local alignment step and patch packaging in Rust. While we acknowledge the two Rust ports bsdiff-rs [52] and bidiff [53], our analysis showed that the patches they generate diverge significantly from those produced by the original bsdiff4. To ensure compatibility and reproducibility, we therefore developed a complete Rust reimplementation of bsdiff4 as a foundation for bsdiff6 that will be released alongside our bsdiff6 port.

Our Rust implementation reproduces identical patches to the original bsdiff4 for most input pairs. Minor discrepancies arise from differences in the suffix-matching logic. The original implementation does not always identify the longest possible suffix, whereas the `sacapart`[5] crate we employ can do so reliably. Consequently, the two versions occasionally outperform each other depending on the input. This behavior stems from the local alignment phase, which skips indices that are part of a region aligned prior. For alignments that are shorter than expected, this can lead to leftover bytes being aligned more or less effectively elsewhere, introducing slight variations in alignment efficiency.

For bsdiff6, performance differences are again minimal but observable. Since the bsdiff6 local alignment step does not skip aligned indices, these differences cannot be attributed to the same local alignment issue. Instead, our findings indicate that this behavior originates in the block alignment phase, where our implementation, using standardized libraries rather than custom code, produces subtly different results for certain inputs. Due to the high abstraction level of the input data in this stage, the difference is difficult to pinpoint precisely. However, our investigation suggests that the differing block alignments do not lead to missed or incorrect matches. Instead, they influence the node scoring in the combination step indirectly by altering the node pool.

For both diffing algorithms, we publish a performance analysis similar to the one shown in the Table I alongside our implementations.

[5]https://crates.io/crates/sacapart