reUpNix: Reconfigurable and Updateable Embedded Systems

Niklas Gollenstede
niklas.gollenstede@tuhh.de
Hamburg University of Technology
Germany

Ulf Kulau
ulf.kulau@tuhh.de
Hamburg University of Technology
Germany

Christian Dietrich
christian.dietrich@tuhh.de
Hamburg University of Technology
Germany

Abstract

Managing the life cycle of an embedded Linux stack is difficult, as we have to integrate in-house and third-party services, prepare firmware images, and update the devices in the field. Further, if device deployment is expensive (e.g. in space), our stack should support multi-mission setups to make the best use of our investment.

With reUpNix, we propose a methodology based on NixOS that provides reproducible, updateable, and reconfigurable embedded Linux stacks. For this, we identify the shortcomings of NixOS for use on embedded devices, reduce its base installation size by up to 86 percent, and make system updates failure atomic and significantly smaller. We also allow integration of third-party OCI images, which, due to fine-grained file deduplication, require up to 24 percent less on-disk space.

CCS Concepts: • **Computer systems organization** → **Maintainability and maintenance**; **Embedded software**.

Keywords: reproducible systems, embedded systems, NixOS

ACM Reference Format:
Niklas Gollenstede, Ulf Kulau, and Christian Dietrich. 2023. reUpNix: Reconfigurable and Updateable Embedded Systems. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589610.3596273

1 Introduction

In the last decade, embedded systems' software transitioned from single specialized firmware programs to complex Linux-based software stacks. Such embedded Linux systems are most popular among developers [17], who appreciate their source-code availability and compatibility with existing software. Here, they can quickly compose a system from existing components, adapt it, and debug it with well-known tools, instead of building up their own development stack, perhaps even being dependent on proprietary development tools.

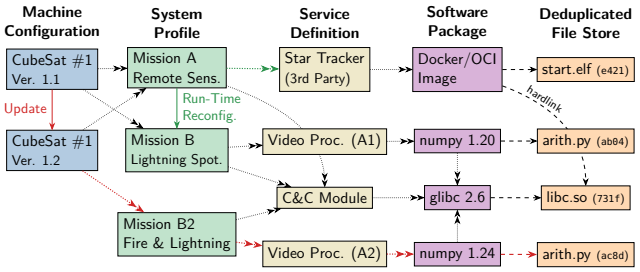

Figure 1. reUpNix Overview. One device has one deployed machine config with multiple system configs, each including different OCI containers and/or Nix services.

However, with those systems also being connected to the network, having software life-cycle management for the whole software stack becomes crucial. We have to answer: (C1) How are third-party and off-the-shelve components *integrated* with our in-house software? (C2) Is the firmware-derivation process *automated* and *reproducible* [22]? (C3) Can *software updates* be applied to deployed systems without endangering the device's functionality? While those questions are already challenging for locally-deployed systems, they become even more critical for remotely-deployed systems as manually reviving their devices is hard.

For example, the "New Space" industry brings more and more Linux systems into space [3, 23, 31], and projects like IBM Endurance [36] or SpaceCloud [27] even aim to provide in-orbit execution platforms for containerized applications. A general challenge for in-orbit updates are high requirements regarding reliability and limited upload bandwidth [40]. The latter mandates efficient processes, as uploads of the entire firmware, especially when it comes to Linux based Payload-Computers, directly conflict with the uplink bottleneck.

Orbital clouds could reduce costs and orbital debris by reusing *existing* hardware, but also more mundane environments like smart-cities would also benefit from sharing deployed sensor networks. This brings forward two more life-cycle challenges: (C4) How can we co-locate *multiple*

LCTES '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0174-0/23/06…$15.00
https://doi.org/10.1145/3589610.3596273

system configurations on the same machine, and (C5) how do we *reconfigure* these systems at run time?

With reUpNix, we propose a methodology for designing Linux-based software stacks for embedded systems that are reproducible, updateable, and allow for reconfigurable multi-mission scenarios (see Fig. 1). For this, we build on the existing NixOS [14] Linux distribution, which is based on the functional Nix package manager [13]. While Nix was already used for packaging reproducible research [20] and HPC workloads [8], it comes with trade-offs and shortcomings that hinder its adoption for embedded systems.

In this paper, we identify and alleviate NixOS' problems and make it more suitable for usage in embedded systems. We consider the contributions of this paper as follows:

- We identify the shortcomings of NixOS for embedded systems and shrink its minimal deployment size.
- We propose a design that integrates multiple system configurations, traditional containers, and isolated Nix-native applications into one machine state.
- We make the machine-update and system-reconfiguration process failure atomic and shrink the update size.

The rest of the paper is structured as follows: In Sec. 2, we describe Nix, NixOS, and its different shortcomings that hinder its adoption in embedded systems. In Sec. 3, we address those issues and propose reUpNix as a methodology to build embedded software stacks. We evaluate the quantifiable aspects of our approach in Sec. 4, discuss our results and reUpNix in Sec. 5, before we compare it to the related work (Sec. 6), and conclude our paper (Sec. 7).

2 Nix and NixOS

Nix [13] is a software-package manager with its own functional configuration language. A Nix program specifies external dependencies (e.g., sources) and configures the build process of multiple, intertwined, and dependent packages. When evaluating these programs, we *derive* per-package build instructions, which we *realize* in topological dependency order by executing the build commands in a sandbox environment without network connectivity. Result of this derivation are per-package file trees; the Nix *components*.

Shortcoming S1 (Language Flexibility): *Nix gives developers a high degree of freedom in organizing their software stack. Without further guidance or design structure, this can result in high maintenance costs.*

The derived components are stored as content-addressed artifacts in the *Nix store*. To derive the address, the Nix interpreter calculates a cryptographic hash over the derived build instructions, including the hashes of all build dependencies and source-code archives used. Due to this transitive capture and Nix's build sandboxing, for a component's content to change, its address has to change as well. Since any (indirect) version update results in a different address, Nix implements *strict version pinning*.

The *Nix store* is a global directory (`/nix/store/`), where components are stored as subdirectories whose names contain their component addresses. As these components are isolated file-trees, all run-time dependencies between components have to be expressed as *absolute* file paths into other components. For example, Nix uses the `rpath` property of ELF binaries, which hard-codes the library-search path, to refer to shared-library components explicitly. Different versions and variants of the same package can thus co-exist on the same machine, and updates to newer versions are transactional and can always be rolled back as long as the old components are not removed. However, the necessity for explicit references also results in component addresses being sprinkled throughout many files. In combination with the transitive hashing, this results in small updates having an amplified change impact.

Shortcoming S2 (Size of System Updates): *As Nix amplifies the change impact, system updates often have to replace components although (parts of) their semantic did not change. In network-constrained (i.e., low bandwidth, uni-directional connection) settings, the update transfer size is problematic.*

While transfer size is usually not a concern for the typical (desktop or server) Nix user, the space requirement of the Nix store is a known issue, and with frequent updates, the Nix store accumulates (outdated) components over time. Therefore, Nix employs two mechanisms to reduce its on-disk size: file deduplication and garbage collection.

For *file-level deduplication*, Nix uses a global content-addressed *links* directory, where each file is stored under its checksum. Whenever a component is pushed to the store, Nix calculates the checksum of each file and queries the *links* directory. If the file exists there, it is hard-linked and not copied to its new location. Thereby, components, while having different addresses, can still share resources on the file level, even if they originate from different source packages. Please note that this deduplication is only possible as all files in the Nix store are immutable and read-only.

To get rid of outdated components, Nix performs component-level *garbage collection*. By scanning files for strings that look like component addresses, Nix calculates the *retained dependency graph*, which is a subset of the build-time dependency graph. Given a root set of currently-used components, unreachable components are garbage collected.

Based on the Nix package manager, *NixOS* is a Linux distribution that extends Nix's component model to cover a complete system configuration. Included in *nixpkgs*, it comes with an extensive package collection that currently consists of over 80 000 package definitions.

Shortcoming S3 (System Size): *NixOS packages usually aim for functionality over minimal output sizes. Further, as the automated dependency-graph scanning over-approximates the actual run-time dependencies, NixOS systems tend to include non-essential functionality and have a large installation size.*

To build a complete system, NixOS stores configuration (files) as components and introduces system profiles, a collection of configurations and packages that, in combination, form a complete, bootable system. At boot time, the user chooses a profile at the bootloader, which loads the kernel and the NixOS initial ramdisk, both of which reside outside the Nix store. The ramdisk mounts the Nix store, and hands over to the profile's system-init process. For services or libraries that require well-known paths (like `/etc/passwd`), NixOS populates `/etc` with symbolic links into the Nix store and files that are dynamically created during boot.

Shortcoming S4 (Bootloader Updates): *While NixOS builds the kernel and ramdisk as components, it does not cover configurable disk partitioning, and bootloader configurations are partially realized outside the Nix build process, relying on run-time system state. Therefore, the boot setup is not inherently reproducible and updating it is not yet transactional.*

While NixOS uses static version pinning to solve the library-version problem, Docker [26], which further isolates services via Linux namespaces, popularized another solution: Docker services come as *open container initiative (OCI)* images, which wrap an application in a complete Linux installation (without a kernel), for which specialized size-optimized distributions, like Alpine Linux [1], were developed. At this point, OCI images can be considered the industry standard for cross-vendor collaboration and service deployment.

Shortcoming S5 (Third-Party Packaging): *While NixOS directly supports running (service) components in Linux namespaces, its current OCI integration requires a separate runtime. Together with the entry burden of the Nix language, Nix only poorly supports the integration of third-party packages.*

Summarized, Nix and NixOS provide some unique features (complete build and deployment description, transactional updates, library multi-versioning), but it also has drawbacks that are especially problematic for embedded systems (system and update size, bootloader updates, language flexibility, and third-party software).

3 reUpNix – A NixOS-Based Design Methodology for Embedded Systems

With reUpNix, we propose a methodology to ease these shortcomings (S1-5) while also addressing the mentioned challenges (C1-5) for embedded Linux stacks.

3.1 Machine-, System-, and Service Configurations

To tame the flexibility of Nix (S1) and to give system designers guidance in structuring their systems, we propose a layered methodology that builds on existing NixOS concepts but restricts and extends them with embedded systems in mind. On this level, we also address the third-party integration (C1) and the multi-configuration (C4) challenge. In the example (Fig. 1), a cubesat carries two missions (remote sensors, lightning detection), which both use the camera hardware; the second mission is then updated to also detect wildfires. We discuss our methodology from the top down:

Machine Configuration The *machine configuration (MC)* describes the state of a computing system (e.g., a Raspberry Pi) at a given point in time. It lists the installed software and its configuration, but also covers the disk partitioning and the bootloader configuration. The MC is defined in a Nix program, and the developer can describe multiple machine states in the same file, which can reuse and refine each other. Thereby, a reUpNix repository can configure a fleet of devices as well as multiple evolution steps per device.

From a derived MC, our automated installer builds a bootable machine image. Up until now, NixOS does not provide an automated install process with flexible partition setups, which is however necessary to get a fully reproducible and automated embedded-system derivation pipeline (C2). Our installer sets up loop-mounted images, performs the partitioning, and copies Nix-store components from the build host to the image. For this, we copy all components that are reachable in the retained dependency graph of the MC. Also, we statically tailor an application-specific boot script that replaces the usual one-size-fits-all boot script of NixOS.

At this level (see Fig. 1), we also see that a machine update, covered in Sec. 3.3, is the transition between two MCs.

System Profile Below the MC level, we use the existing *system profile (SP)* abstraction of NixOS to describe a bootable system state. Each SP consists of the reUpNix *base system*, hardware-specific components (e.g., loadable kernel modules), and user-defined *services*. Each MC explicitly references the set of SPs (C4) that should be deployed at one point in time (by installation or update).

SPs also address the third-party integration problem that is specific to embedded systems (C1): In the cloud, the Docker model became successful as it assumes that hardware is essentially uniform and can be reduced to the file system and network. However, for embedded systems, this assumption does not hold, as applications often require specific hardware drivers, which have to be inserted into the host kernel, breaking usual isolation models. By putting hardware-specific components at the SP level, we acknowledge that system integrators sometimes have to break service isolation and different profiles can use different drivers. Further, instead of making a complete container *privileged* (as Docker does), reUpNix demands that the integrator clearly defines how to integrate those privileged third-party modules.

Also at the SP level (see Fig. 1), we define that *run-time reconfiguration*, which we will also cover in Sec. 3.3, is the transition between two system profiles.

Services Each SP lists the services that should be started at boot. Besides the base system and hardware-specific components, reUpNix requires everything else to be a service. All reUpNix services are started as containers in their own Linux namespace, where components from the Nix store and

a minimal container base system are visible. Instead of reinventing the wheel, and since we already include `systemd` [30] in our base system, use `systemd-nspawn` to start and manage services at run time. `systemd`'s nested system management, across container boundaries, gives us a complete view of the dynamic system state at the booted SP level.

We support two types of services: Nix and OCI services. With *Nix services*, a NixOS standard functionality that was described as "graph-based containerization" [19], a regular Nix component becomes the entry point for a service container.

For OCI services, we integrate standard OCI images [25], which come as collections of file-system *layers*, into the system (C1). Usually, the container runtime (e.g., Docker) assembles these layers with `overlayfs` into a file-system tree for the container. Layers are also the granularity of reuse. If, for example, two images are derived from the exact same Alpine distribution, the Alpine base layer is stored only once.

For reUpNix, we use a fine-grained sharing method: we collapse all layers of an imported OCI image into one directory and store it as a Nix component. To start the services, we use this component as the root directory for the container and use `overlayfs` to provide, like Docker, a writable upper layer. As the Nix store performs deduplication at the file level, files can be reused between OCI images, but also between the base system, Nix services, and other OCI services. Also, as a result of collapsing, deleted files are truly removed from the system and not only marked as deleted in a higher layer.

In our example (see Fig. 1), the "star tracker" for the remote sensing mission is included as an OCI image. As the image also uses `glibc 2.6`, the Nix store shares the `libc.so` file between the image and the regular Nix component.

3.2 Size Reduction of the Base System

As NixOS was intended as a desktop and server Linux distribution (S3), where disk space is abundant, it is rather relaxed when it comes to disk usage. For embedded systems, a large system not only requires a larger storage medium, but it also results in larger and more frequent updates (S2) as the update "attack surface" is enlarged. Therefore, for reUpNix, we made an active effort to shrink the size of the base system included in every MC. For this, we (1) remove standard components and (2) shrink the size of individual components. Instead of discussing all of our modifications, we describe our minimization procedure exemplarily, naming our principles.

Localization Many packages include additional, non-executable files that are not necessary for the system-level core functionality. For example, to be usable for end users, NixOS includes language localization for basic components by default. As error messages will not be end-user visible, we used existing NixOS configuration options to remove localization from the system. *Principle*: Use existing feature flags to shrink components to their functional core.

Perl interpreter By default, every NixOS installation includes the `perl` language interpreter as it is used to dynamically populate different configuration files (e.g., `/etc/passwd`). To get rid of this dependency, reUpNix creates the `/etc` as a read-only store component, with more files pre-generated, and optionally uses an `overlayfs` to provide for dynamic modifications (i.e., secret injection). By making this NixOS run-time configuration step static, we could remove `perl` as a run-time dependency. *Principle*: Move dynamic variability to the derivation time.

Table 1. Qualitative Comparison of the Update Strategies

	complete	efficient	reprod.	safe	reconfig.
In-Place Destructive [6]	++	+	- -	- -	- -
Recovery OS [5, 38]	+	-	+	+	- -
A/B Partitioning [5, 35, 38]	++	- -	+	++	(+)
Min. OS + OCI [11, 37]	- -	+	○	+	+
Merged Trees [2, 14]	+	++	++	+	++

Linux kernel NixOS, like most [33] distributions, compiles almost all drivers as loadable kernel modules to be prepared for dynamically attached hardware. However, for embedded systems, the hardware configuration is more stable and predictable. Therefore, we used `localmodconfig`, with some manual overwrites, to enable only those drivers that are necessary for each platform. Further, more elaborate methods based on dynamic tracing could be used in the future [21, 33]. *Principle*: Statically specialize software for the combination of hard- and software.

3.3 Atomic System Reconfigurations and Updates

As already discussed, updates (C3) are a critical point in the life cycle of an embedded system, especially if it is deployed in an inaccessible environment. Therefore, we have to ensure that the machine always remains in a recoverable state and that updates cannot "brick" the device. Before we detail our approach, we assess different update strategies for Linux devices (see Tab. 1) in five dimensions: A *complete* strategy can update all system components, and it is *efficient* if small changes induce small updates. A *reproducible* strategy forces the system to a defined state, and a *safe* one never leaves the system in an unusable state. *Reconfigurable* strategies allow for shared multi-mission systems.

Traditional Linux package managers (e.g., Debian's `dpkg`) adhere to the Linux *file-system hierarchy (FHS)* [34], where every file has exactly one place in the file system. Therefore, updates have to be *in-place and destructive*, which provokes inconsistent intermediate states that can become permanent on errors (e.g., power outage) [14]. Further, they are not reproducible (- -) as the outcome depends on the initial system state, which becomes problematic if the gap between the system and the upstream distribution grows larger [12].

To make over-the-air updates for mobile devices safe, current systems, like Android [38], employ a *recovery OS* and/or an *A/B partitioning* scheme. For this, the system is split into a read-only image and a writable data partition. For the update, a new read-only image is installed either by the recovery OS, which is a minimal fall-back OS, or by the current OS

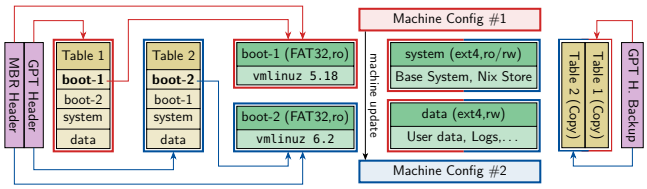

Figure 2. Partition layout with two Boot Slots

onto the inactive partition. While both methods are safe (+), and A/B partitioning even ensures a fully-operational device (++), they both require the transfer of whole system images (efficiency: -). Further, A/B partitioning doubles the storage requirement for the system image, which however also provides limited potential for multi-mission scenarios.

Another potential road for embedded systems is a *minimal base system with a container runtime* [11, 37]. By exchanging containers, the system can be reconfigured for other missions, but there is no strategy to update the base system. And while the update of one container is reproducible (○), embedded applications often cannot be fully containerized (kernel modules), and thus still depend on the base system.

NixOS [14] and OSTree [2] use a merged-tree approach, where multiple system profiles live in the same file system, which enables transactional and reproducible system updates. In contrast to NixOS, OSTree does not break the FHS by using extensive hard linking to create a standard view from the component repository. However, both methods (S4) do not cover the bootloader and require a writable yet failure-safe file system (e.g., by journaling).

reUpNix Updates With reUpNix, we combine the merged-tree approach of NixOS with A/B partitioning for the bootloader, and use containers for third-party components. For this, we will first look at the partition setup (Fig. 2).

First of all, reUpNix uses a single system plus one data partition, which all MCs and SPs share. While the data partition is generally writable, the system partition is only mounted writable in a separate namespace for the update process. For the boot partition, which contains the bootloader (configuration), kernel(s), and initial ramdisk, we use A/B partitioning. We thus maintain the reconfiguration and sharing properties of the merged-tree approach with the completeness and robustness of A/B partitioning. Please note that updates are prepared offline at the build host (covered in Sec. 3.4).

As already mentioned, NixOS does not cover the boot-partition files as regular Nix components. For reUpNix, however, we derive them (C2) within the regular reproducible and automated build process. To perform MC updates, the update script creates a new FAT32 file system in partition B, copies all files from the Nix store, and switches A and B. Afterwards, the boot partition A contains kernels and initial ramdisks for all SPs of the new MC. By *not* storing a complete boot-partition image, we can reuse files between MCs and reduce the size of updates.

To switch between the A/B boot partitions in a generic way, without relying on features of any particular firmware early-stage bootloader, we duplicate the GPT partition tables. Each table lists all partitions, but with a different boot partition as the first / EFI-system / MBR-bootable partition. By rewriting the GPT *header*, and if it matters for the device also the MBR, we can switch between the boot partitions. For this, we only have to overwrite the first two 512-byte sectors of a flash medium, instead of rewriting the whole GPT partition table. As flash storage can only write entire flash pages (usually 4 KiB), they provide write atomicity on this granularity [32]. Therefore, reUpNix's boot-partition switch is atomic on such devices.

In total, we perform these steps for an MC update: (1) insert new components into the Nix store, (2) update the inactive boot partition, (3) switch A and B partition tables, (4) reboot to a new SP, and (5) remove unnecessary components from the store. With this process, and under the assumption that adding files to the system partition does not corrupt its state, we ensure failure atomicity for the update process.

Besides MC updates, reUpNix also supports run-time reconfigurations by rebooting into a different SPs. At this point, we decided to support only this clean-boot approach for system reconfiguration, as dynamic run-time reconfigurations, where services are stopped and started, is more risky and does not support all reconfigurations (e.g., different kernels). However, in future work, we want to look at safe no-reboot reconfigurations as they promise shorter downtimes.

While we require that a MC's default SP, which should only contain a base system, is bootable, we can recover from broken non-default SPs. For this, we use a hardware watchdog and one-shot bootloader overrides, so that we can try booting into other SPs without ever bringing the device into an unsafe state. We support this on UEFI (x86 and Aarch64) systems with `systemd-boot` and U-Boot-based systems (Raspberry Pi and NXP i.MX 8M+). To support one-shot boots with U-Boot, we create a temporary U-Boot environment that selects the intended boot entry, erases itself, and then boots the SP. If booting fails, we fall back to an MC-specific U-Boot environment that boots the default SP.

3.4 Transfer Size of Updates

With our reUpNix design methodology and our update/reconfiguration strategy in place, we now want to look closer at step (1) and step (5) of our update mechanism. reUpNix supports uni-directional updates, where an update archive is prepared upfront and sent to the device without the necessity for a back channel. In contrast to bi-directional update methods (e.g., based on `rsync`), this is more suitable for high-latency networks (e.g., satellites) and large device fleets.

As already mentioned, an update transitions a device from the current MC to a new MC. For this, we have to install the transitive dependency closure of the new MC to the device (step 1) and remove those components that are only

```
L1 # Step (1): ____ Source(s) _______   ____ Destination File __________
L2 L /nix/../a5c0-numpy-1.20/arith.py   /nix/../d221-numpy-1.24/artih.py
L3 C /nix/../bar              128-768   /nix/../foo    256 # reuse block
L4 B ${UP}/blocks             256-512   /nix/../foo    896 # new block
L5 P /nix/../bin/bash ${UP}/bspatch.1   /nix/../bin/bash   # patch file
L6 # Step (5)
L7 R                    /nix/../a5c0-numpy-1.20 # remove component
```

Listing 1. reUpNix Update Commands

reachable from the old MC (step 5). With `nix-copy-closure`, NixOS already comes with a standard method to transfer the transitive closure to a different machine. However, this method is bi-directional, and it always transfers complete components, which further inflates update-transfer size (S2).

To make reUpNix suitable for embedded devices, we explore different update-compression methods that provide small transfer sizes for whole MCs, while keeping the required resources for applying the update on the target device in mind. Our lever for shrinking updates is that our target device already has a Nix store that is populated with the old MC. Our preparation results in an update script that the vendor can transfer over a uni-directional connection.

Update Script We define a custom format for the update script, which interleaves commands and newly-introduced data blocks, and which we compress with *zstd*. Due to the interleaving, the update is streamable, and we do not need to buffer it prior to its application. Only for binary patches we require additional space for storing one patch. For easier understanding, we present our example update (Lst. 1) as if new blocks were extracted to `${UP}` and referenced by name.

File Deduplication Instead of sending complete components, we transfer only those files which do not yet exist in the target device (see Fig. 3 a). For this, we can use the deduplication feature of the Nix store to find files that only exist in the new MC. On the technical level, we build up a bi-directional inode–path mapping of the transitive closure for both MCs. For a file that is already on the device, the update script only creates a hard link (see Lst. 1 L2). As this increases the inode's reference count, we can delete the old, now obsolete, file reference (L7).

Partial File Reuse However, as component addresses are scattered throughout many files (S2), the file deduplication mechanism will find fewer equal files than it would on other Linux distributions. Therefore, we want to reuse parts of files that are already on the device to further drive down the update size. For this, we explore two methods: global block reuse and `bsdiff` [28, 29].

For global block reuse (see Fig. 3 b), we chunk all files of the old MC into fixed-sized blocks and create a hash index over those. On the non-duplicated files in the new MC, we perform the same chunked hashing to find blocks that are already on the device. After collapsing consecutive blocks from the same source file, we extend the update script with a *copy block* command (L3), which copies parts of the source file (`bar`) to create parts of a new file (`foo`).

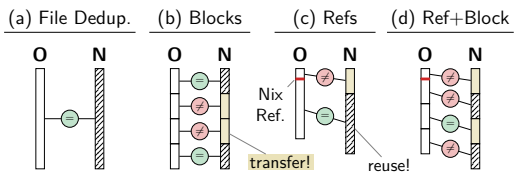

Figure 3. In-File Block Patterns to reconstruct a **N**ew file from an **O**ld file that is already on the target machine.

The scattered component address, which can occur at any offset, pose both a challenge and an opportunity: If an addresses change, they force the enclosing block to be transferred in the update. However, they are also visible anchor points in the file to identify content that has shifted by non-block–sized increments. Therefore, we refine our fixed-size chunking: (1) We scan files for store references and split them around those boundaries (see Fig. 3 c). (2) The resulting segments can be further split and reused with the previously described fixed chunking (see Fig. 3 d).

BSDiff We further explore the usage of `bsdiff` [28, 29] to compress the transferred size. While we assume that the build host has a large amount of memory, the target device's memory is often limited. Applying a binary patch, however, requires enough memory to hold the old and the new data, as well as the *uncompressed* patch. Therefore, it is not possible to use `bsdiff` at the MC level, but we have to apply it at the component or file level.

For `bsdiff` to create small patches, we have to invoke it on pairs of similar artifacts (e.g., components, files, …), where one of them already exists on the target while the other is about to be installed. As finding the smallest possible patch would require an exhaustive patch generation with every existing artifact, we require a heuristic method to find similar pairs. Usually, with a traditional package manager, we would simply pair up the previous package version with the current one. However, with Nix, the problem is more complex, as (a) components are primarily hash addressed, (b) multiple package versions can co-exist, and (c) the same definition can be instantiated with different options. Therefore, there is no unambiguous pairing.

We tackle this problem by matching subgraphs of the retained dependency graph that are reachable from both MC components. To avoid the NP-complete subgraph isomorphism problem, we further label the dependency edges with the component's derivation name,[1] which, however, can be ambiguous, and perform a heuristic match operation: Initially, we merge both MC components into a pair node. Recursively, we *greedily* pair components together if they are reachable from an existing pair by a locally unambiguously-labeled pair of edges. For unmatched components, we fall back to file deduplication with optional block reuse.

[1]Nix includes a derivation name and a version string in the component path. For example, for …`a14bf-glibc-2.6`, we use `glibc` as the label.

Table 2. Overview of Base-System Storage Requirements

		Disk Use [MiB]		Nix Store			
		Boot	System	[MiB]	ELF Sz.	Comp.	Files
x64	NixOS	18.2	1072	1023	47.44 %	747	23042
	reUpNix	14.1	141	134	74.49 %	325	3567
Aarch64	NixOS	55.5	1092	1040	48.74 %	751	24033
	reUpNix	51.8	184	176	67.99 %	323	3994

For matched components, we invoke `bsdiff` on deterministically created component archives, while for files we pair those with the same intra-component path. We also support block-based invocation of `bsdiff`: Within paired files, we create binary patches of blocks with the same index, which, if necessary, we re-align at hash-identical blocks. The `bsdiff` patches receive their required compression with the update script. In the update script, we invoke `bspatch` with a source component/file/block, a patch, and the target component/-file/block (L5).

4 Evaluation

For our evaluation, we compare reUpNix on two platforms: (1) x64/AMD64 with UEFI boot and (2) Aarch64 with the U-Boot bootloader on a Raspberry Pi 4. They represent typical systems in the embedded area and future applications in the New Space [16]. We will characterize the base systems, investigate the costs of service integration, compare transfer-size reduction mechanisms, and look at the reconfigure time.

4.1 Base System

First, we will compare the installation size of a standard NixOS system to our minified reUpNix system on the two evaluation platforms. For this, we build a base system with a `bash` shell, a `dropbear` SSH server, the container infrastructure, and the ability to receive updates via network.

In Tab. 2, we compare the disk usage of the four base systems and characterize their Nix store. With our systematic minimization, we could reduce the overall installation size (both partitions) for x64 by 86 percent and by 79 percent for Aarch64. To achieve this, we eliminated more than half of all components. The biggest contributor to the sharp decrease in deployed files is the Linux kernel, where alone we removed 7160 files for Aarch64; mostly kernel modules and device-tree files that are not required on our Raspberry Pi.

From the largest store components (see Tab. 4), we see that language localization, which is not necessary on embedded devices, inflates the NixOS base system. Further, we could even remove `nix-2.8.1`, which contains the Nix package manager, from both installations. This is only possible as all changes to the on-device store are prepared off-device on a separate build host and are injected via updates. Also, for the Linux kernel, our device-specific tailorization is more successful for x64 than for Aarch64, since `localmodconfig` only removes modules, but the standard NixOS kernel for the Raspberry Pi 4 enables many features statically.

Table 3. Storage Cost of Individual Services

	OCI Image		Additional Components	
[MiB]	Smallest	Largest	Δ NixOS	Δ reUpNix
httpd	54.6	142.5	8.6	79.8
mariadb	— 371.9 —		233.9	303.8
memcached	8.0	88.4	0.5	1.3
mongo	— 668.3 —		167.6	205.8
mysql	436.2	510.1	404.9	580.7
nginx	23.9	139.2	19.1	24.1
node	162.4	967.9	50.0	139.1
postgres	209.5	366.2	38.3	168.9
python	49.3	900.5	23.1	23.1
rabbitmq	123.6	223.4	431.2	472.1
redis	28.3	114.8	5.5	5.5
registry	— 23.5 —		28.3	29.7
traefik	— 102.8 —		102.0	103.4
wordpress	— 593.4 —		423.4	516.7

Another interesting case is `extra-utils`: Nix detects this package *sporadically* as a retained dependency of the initial ramdisk, which indeed contains the address of that component but never uses it (an expected over-approximation). However, the address is only discovered sporadically as the initrd gets compressed, whereby, depending on the compression result, the address becomes invisible to the dependency scanner. Therefore, the Nix dependency scanner is only sound if the component addresses are not opaque.

Summarized, reUpNix provides a minimized base system that is able to execute OCI images and Nix services in isolated Linux namespaces.

4.2 Size of Nix and OCI Services

Next, we will look at the integration of OCI images into reUpNix. For this, we will compare the static storage costs of OCI and Nix services, as well as the benefit of having file-level deduplication of OCI files. As base for this comparison, we selected the top-15 recommended x86 Docker images from dockerhub.com and their variants.[2]

Nix Service vs. OCI image First, we look at the static storage costs that are induced by a third-party supplied OCI image in comparison to a Nix-native service, which can reuse components also used by the base system (e.g., glibc). For each service, Tab. 3 shows the largest and the smallest Docker image, while five images only provide a single variant. We also add the same services as native Nix packages to both NixOS and reUpNix, to show the combined size of the *additional components* that are not yet part of the base system. For this comparison, we excluded the golang container, as it only includes a compiler and no service-related binary.

We see that installing a service (except redis) on reUpNix requires more additional components than on top of standard NixOS, as fewer components are already part of reUpNix's small base system. For eight services, reUpNix requires less

[2] **OCI Images:** golang (1.19.0, dAba), httpd (2.4.54, dba), mariadb (10.8.3, d), memcached (1.6.16, dba), mongo (5.0.10, d), mysql (8.0.30, dA), nginx (1.23.1, da), node (18.7.0, dASba), postgres (14.5, dba), python (3.10.6, dASba), rabbitmq (3.10.7, da), redis (7.0.4, dba), registry (2.8.1, d), traefik (2.8.3, d), wordpress (6.0.1, d) **Variants: d**efault, **A**lternative, **S**lim, **b**ullseye, **a**lpine

Table 4. Largest Components. Deleted (×), minimized (↓).

	x64/UEFI				Aarch64			
Top	NixOS		reUpNix		NixOS		reUpNix	
1.	× glibc-locales-2.34	214.8 MiB	extra-utils	15.4 MiB	× glibc-locales-2.34	214.8 MiB	↓ linux-5.15.56	74.0 MiB
2.	↓ linux-5.15.56	105.2 MiB	↓ linux-5.15.56	14.5 MiB	↓ linux-5.15.56	137.8 MiB	↓ systemd-250.4	14.2 MiB
3.	× perl-5.34.1	52.7 MiB	↓ systemd-250.4	13.6 MiB	× perl-5.34.1	52.5 MiB	↓ glibc-2.34	8.1 MiB
4.	↓ systemd-250.4	40.1 MiB	↓ glibc-2.34	9.8 MiB	↓ systemd-250.4	41.7 MiB	util-linux-minimal-2.37.4	6.4 MiB
5.	↓ glibc-2.34	39.2 MiB	↓ initrd-linux-5.15.56	6.7 MiB	× icu4c-71.1	35.7 MiB	↓ initrd-linux-5.15.56	5.7 MiB
6.	× icu4c-71.1	36.0 MiB	gcc-11.3.0-lib	6.1 MiB	↓ glibc-2.34	35.6 MiB	gcc-9.3.0-lib	5.5 MiB
7.	× icu4c-67.1	33.8 MiB	util-linux-minimal-2.37.4	6.1 MiB	× icu4c-67.1	33.5 MiB	shadow-4.11.1	3.9 MiB
8.	× spidermonkey-78.15.0	32.4 MiB	openssl-1.1.1q	4.0 MiB	× binutils-2.38	32.0 MiB	db-5.3.28	3.7 MiB
9.	× binutils-2.38	31.2 MiB	db-5.3.28	4.0 MiB	× spidermonkey-78.15.0	31.2 MiB	openssl-1.1.1q	3.7 MiB
10.	× nix-2.8.1	15.3 MiB	shadow-4.11.1	3.9 MiB	× nix-2.8.1	13.9 MiB	iproute2-5.17.0	3.5 MiB

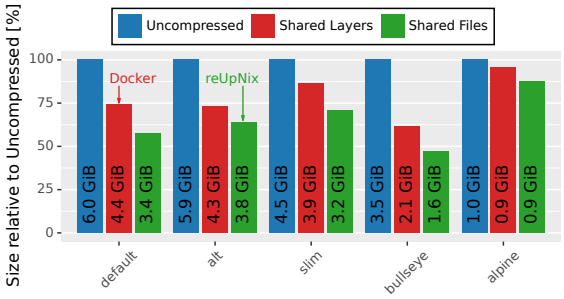

Figure 4. Combined Storage Cost of OCI Images

space than the smallest OCI image (green), which highlights the potential benefit of a well-crafted Nix package. Of these, the commonly used key-values stores redis and memcached come with a particular small footprint and require only 20 resp. 17 percent of the smallest OCI image.

For four packages, the Nix service requires more space than the largest OCI image (red). Due to many retained dependencies in the NixOS package, RabbitMQ comes with a tripled installation size. As we have seen with the base system, though, Nix package can be systematically debloated (see Sec. 3.2). Technically, there is no reason for a reUpNix service to be larger than an OCI image.

File Deduplication vs. Layer Sharing With reUpNix, we collapse the layers of OCI images and import them into the Nix store, which results in file-level deduplication. To measure the benefits of this approach over the usual layer-reuse approach of Docker [26], we import multiple OCI images into the same Nix store, and compare its size to the combined size of the deduplicated layers (see Fig. 4). Further, as the Docker registry supports multiple variants per OCI image, we compare different combinations of image variants.

When we co-locate all *default* images, we see a 22 percent improvement over the already efficient layer-sharing. When using the *alternative* variants (where available), the non-shared layers become more diverse and can thus share fewer files. With the *slim* variants instead, it reduces the uncompressed size, but also largely removes shared files and layers, thus reducing the efficiency of both sharing approaches.

For the Debian *bullseye* and the *alpine* bars, we subset our 15 services to those that have a variant based on the

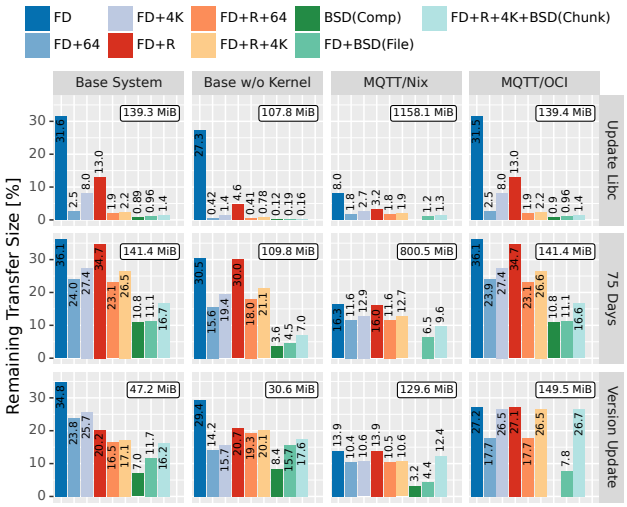

Figure 5. Update-Transfer Compression. We deduplicate files (FD), split at fixed blocks (64, 4K) and at Nix references (R), and use `bsdiff` (BSD) at different granularities.

respective base image. For *bullseye*, where the large base image imposes a significant sharing opportunity, both methods reduce the on-disk overhead. Nevertheless, file-level deduplication saves another 24 percent for bullseye. For Alpine, where the common base image is already much more condensed, we can save only 8 percent, though that is still 1.81 times more than layer sharing achieves here.

Summarized, Nix services often require less disk space than the smallest available OCI image. But even if the system integrator uses a third-party OCI image with reUpNix, the per-file deduplication within the Nix store allows for fine-grained sharing between different unrelated containers.

4.3 Update-Transfer Size

To quantify the update-transfer size and compare our compression strategies, we look at multiple MCs and apply multiple updates with different semantics (see Fig. 5). For this, we focus on the x64 variant of reUpNix.

We apply our updates to empty reUpNix base systems, without any installed services, and to systems with an MQTT

broker and a Zigbee-to-MQTT bridge,[3] which mimics a typical wireless-sensor bridge. For the base systems, we look at the complete base system and at the system excluding its kernel and initrd. For the MQTT systems, we use the same base system, with the NixOS package for Mosquitto and Zigbee2MQTT added in one version, and the services' publicly available OCI images added to the other.

Onto those four systems, we apply three different updates: (1) We introduce a trivial change into the GNU C library that changes its component address but leaves its semantics untouched. (2) We apply an update worth 75 days of NixOS package changes. (3) We update individual packages: `systemd` for the base variants; both Mosquitto and Zigbee2MQTT for the MQTT variants.

Baseline For these system–update combinations, we use the size of the changed components (white boxes in Fig. 5) as our baseline, as the existing `nix-copy-closure` mechanism would transfer exactly those. For the libc and the 75-day update, this naive mechanism replaces (almost) the whole base system (around 140 MiB). Also, we see that the MQTT system that is built only from NixOS packages suffers from large update sizes (S2). Similar to the discussed RabbitMQ package, the NixOS Mosquitto package is much larger than the Mosquitto OCI image.

File Deduplication By removing duplicate files (*FD*), we already improve the transfer size by at least 64 percent. When updating the libc on the MQTT/Nix system, deduplication even reduces the update by 92 percent as the baseline transfers many identical Javascript files (for Zigbee2MQTT).

Fixed- and Variable-Sized Chunking Next, we look at the different chunking methods and compare small and large blocks (*64* bytes and *4K*iB) as well as reference-splitting (*R*). For these variants, we apply chunk deduplication only after removing duplicated files from the update. From the results, we make the following observations and conclusions: (1) small chunks are better than large chunks, even though their greater number could inflate the update script many instructions. (2) reference-splitting alone is rarely better than fixed-sized chunking; if used, it should be combined with chunking. (3) chunking makes NixOS updates efficient, such that small changes (libc) result in small updates.

BSDiff With `bsdiff`, we compare three variants: Either we create a binary patch at the component level, at the file level, or at the block level. First, we see that, for three updates, `bsdiff` is unable to create patches at the component level – it runs into memory corruption issues for too large objects. However, in cases where we were able to derive an update, the component-level binary patches yield the best results and reduce the transfer size between 89.17 and 99.88 percent.

By reducing the granularity of `bsdiff`, we reduce its complexity, but also remove its ability to reuse data. Therefore, the *FD+BSD(File)* variant is less successful than the component variant, but it often beats the block-based approaches. For the considerably-large 75-day update of the MQTT/Nix update, it improves on the best block-reuse approach by another 44 percent. For the MQTT/Nix system, it reduces the libc update from 1158 MiB to 74.9 MiB.

When reducing the `bsdiff` granularity further, down to the block level, the compression rate decreases even more. We do not consider this a viable option.

Summarized, reUpNix is able to considerable compress the update-transfer size of NixOS-based systems, alleviating S2. As compression methods, block-reuse with small chunks and (component-)/file-level binary patches proved to be most successful. We will discuss this further in the next section.

[3] Mosquitto MQTT Broker v2.0.14, Zigbee2MQTT v1.25.0

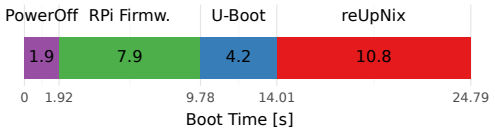

Figure 6. reUpNix Reconfigure Time by Reboot

4.4 Reconfigure Time

As reUpNix switches between SPs by rebooting, we want to quantify the time it takes to reboot into SPs. For this, we set up a Raspberry Pi 4 two SPs, co-located on a micro-SD-card, that store the two MQTT/Nix variants used for the Version Update experiment from Sec. 4.3. We repeatedly reconfigure the system from the old to the new SP, while timestamping the serial output of the Pi. We thus mimic the actual downtime induced by this update, as the patch application can be done concurrently to the normal operation.

The average of the 20 reboots is broken down into individual phases in Fig. 6. From the total reconfigure time of 24.8±0.08 s, reUpNix shutdown (8 %) and bootup (43 %) take make up for only half of the time. The other half stems from the firmware initialization (32 %) and the boot loader (17 %), where only the latter depends on reUpNix as it loads the kernel (1.98±0.0024 s) and the initrd (0.70±0.0005 s).

Overall, we see that reUpNix's current clean-boot approach requires a significant time span. While this is acceptable for MC updates, we want to improve on this in the future for reconfiguration. Here, the use of `kexec()` is a possible path to avoid the bootloader and firmware latencies.

5 Discussion

We argue that reUpNix brings some unique features, like minimized updates and multi-mission setups, to the embedded Linux stack. Thereby, reUpNix alleviates the shortcomings of NixOS for the embedded domain on the conceptual and the technical level. Further, reUpNix, and its base Nix, have more benefits and interesting aspects in this domain.

System-Update Compression As shown (Sec. 4.3), `bsdiff` at the component level produced the smallest updates, but also crashed on large components. Further, this

method also requires holding both components as well as the uncompressed patch, which is usually larger than the destination component, in memory. Therefore, this coarse-grained method should only be applied for large embedded systems with sufficient memory or when limiting its usage to small components. For medium-ranged embedded systems, we recommend the combination of file deduplication and file-level binary patches, as it limits the resource consumption during update application on the target.

For memory-scarce systems, we consider the block-reuse method with small blocks the best option, as it provides small updates but does not need to execute complex algorithms on the target device. Combined with our streamable update script format, this provides a good trade-off between transfer size and on-device resource requirements.

Traceability By design, Nix-based systems already lean towards being more reproducible, as we capture the complete build and integration process within an executable description, whose reproducibility can be validated by re-execution. Further, we could use the whole methodological toolbox of static analyses to connect deployed artifacts to their Nix definitions. Such end-to-end traceability, from the source-code line to the ELF section, would foster trust in the build process and could help to identify update-induced run-time anomalies.

Secure Boot With secure boot, the vendor establishes an unbroken trust chain from the bootloader, over the kernel, to the stored files. Often, embedded secure-boot chains use the `dm-verity` kernel module, which ensures file-system integrity by checking signatures on the disk-block level. Though this avoids on-device key management, `dm-verity` requires a read-only partition.

As this is incompatible with fine-grained updates, reUpNix would require a different file-system integrity schema. Nevertheless, since reUpNix treats files as immutable, this is much simpler than for systems with destructive in-place updates. One possible route we want to explore in the future is to establish trusted-boot chains on component granularity and on the retained dependency graph. Thereby, different trusted boot chains could exist on the same device, one for each system profile.

6 Related Work

Embedded Linux Toolchains *Yocto Linux*[24] is an automated build tool for embedded system images that uses layered system composition, similar to OCI layers but at build time. *Buildroot* has goals similar to Yocto's, but uses a simpler, more opinionated structure and Kconfig-backend configuration schema. Both toolchains do not support multi-mission setups and require an external update mechanism, such as those discussed in Sec. 3.3 and Tab. 1.

Similar to *SkiffOS* [37], which is an embedded container runtime based on Buildroot, we also argue that the system design and update mechanism must go hand-in-hand. However, by relying only on OCI containers, SkiffOS does not allow for tightly-integrated lightweight services, which reUpNix provides with Nix services. Nevertheless, by having the ability to integrate OCI containers, reUpNix tackles the general criticism with NixOS that it breaks the FHS [37].

Embedded Updates Dong et.al. [15] reduce the update size of embedded sensor nodes by manipulating the firmware binary for higher similarity with the old on-device image. *Feedback Linking* [39] is a similar approach but at the linker level. Bogdan et.al. [7] highlight the importance of the update size for automotive ECUs due to the limited CAN bandwidth, and they propose reusing the old firmware to reduce the update size. In contrast, reUpNix leaves files untouched, and provides reproducible updates of full Linux systems and a safe update path.

Courgette [4] is used to shrink differential Chrome updates by applying `bsdiff` [29] on the disassembled binary. In future work, we want to explore the usage of Courgette for reUpNix, even though it introduces a (re-)assembler as an additional run-time dependency.

Generalization from NixOS Guix [9] is another functional package manager for Linux systems that is very similar to Nix, with Guix System as its Linux distribution. In contrast to Nix, which uses different languages (Nix, bash) for composition and build instructions, Guix uses Scheme as an embedded domain-specific language for both. Also, Guix includes a method to authenticate new Git revisions [10] using signed commits. Nevertheless, as reUpNix does not depend on Nix-language specifics, our design methodology and update path could also be executed with Guix.

7 Conclusion

In this paper, we presented reUpNix, a NixOS-based methodology to describe and derive embedded Linux software stacks. reUpNix is able to integrate third-party OCI images, provides automatable and reproducible system images, and allows for transactional uni-directional updates. Further, reUpNix allows for multi-mission setups with dynamic reconfiguration via system reboots. Compared to NixOS, reUpNix has a 86 percent smaller base system on x64, requires significantly smaller updates (up to 99.88 %), and requires less disk space to store multiple OCI images (up to -24%). Overall, reUpNix brings the benefits of Nix(OS) to embedded Linux systems while we ease the existing shortcomings of Nix. reUpNix is publicly available [18].

Acknowledgments

We thank our anonymous reviewers for their work. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536. Further, some motivation and parts of the content were prepared in the context of the AuRelia project.

References

[1] [n. d.]. *Alpine Linux.* https://alpinelinux.org/

[2] [n. d.]. *OSTree – Git for operating system binaries.* https://ostreedev.github.io/ostree/

[3] 2020. *SpaceX: We've launched 32,000 Linux computers into space for Starlink internet.* https://www.zdnet.com/article/spacex-weve-launched-32000-linux-computers-into-space-for-starlink-internet/

[4] Stephen Adams. 2009. *Courgette.* https://www.chromium.org/developers/design-documents/software-updates-courgette/

[5] Stefano Babic. 2021. *SWUpdate: software update for embedded system.* https://swupdate.org

[6] David Blackman. 2000. Debian Package Management, Part 1: A User's Guide. *Linux Journal* 2000, 80es (2000), 12–es. https://doi.org/10.5555/364352.364661

[7] Daniel Bogdan, Razvan Bogdan, and Mircea Popa. 2016. Delta flashing of an ECU in the automotive industry. In *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI).* 503–508. https://doi.org/10.1109/SACI.2016.7507429

[8] Bruno Bzeznik, Oliver Henriot, Valentin Reis, Olivier Richard, and Laure Tavard. 2017. Nix as HPC Package Management System. In *Proceedings of the Fourth International Workshop on HPC User Support Tools* (Denver, CO, USA) *(HUST'17).* Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3152493.3152556

[9] Ludovic Courtès. 2013. Functional Package Management with Guix. arXiv. https://doi.org/10.48550/ARXIV.1305.4584

[10] Ludovic Courtès. 2023. Building a Secure Software Supply Chain with GNU Guix. 7, 1 (2023). https://doi.org/10.22152/programming-journal.org/2023/7/1

[11] Alison Davis. 2016. Introducing resinOS 2.0. https://www.balena.io/blog/introducing-resinos/ Accessed 2022-08-20.

[12] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. 2008. Package Upgrades in FOSS Distributions: Details and Challenges. In *Proceedings of the 1st international workshop on hot topics in software upgrades.* 1–5. https://doi.org/10.1145/1490283.1490292

[13] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of LISA '04: Eighteenth Systems Administration Conference*, Vol. 4. 79–92. https://www.usenix.org/event/lisa04/tech/full_papers/dolstra/dolstra_html/

[14] Eelco Dolstra and Andres Löh. 2008. NixOS: A Purely Functional Linux Distribution. In *ACM SIGPLAN international conference on Functional programming (ICFP).* 367–378. https://doi.org/10.1145/1411204.1411255

[15] Wei Dong, Chun Chen, Jiajun Bu, and Wen Liu. 2014. Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems. *ACM Trans. Sen. Netw.* 11, 2, Article 22 (jul 2014), 34 pages. https://doi.org/10.1145/2629479

[16] Teledyne e2v Semiconductors. 2023. Radiation Tolerant Quad ARM Cortex A72. https://semiconductors.teledyneimaging.com/en/products/processors-and-processing-modules/ls1046-space/

[17] EETimes, Aspencore. 2019. 2019 Embedded Markets Study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[18] Niklas Gollenstede and Christian Dietrich. 2023. *reUpNix: Reconfigurable and Updateable Embedded Systems – https://github.com/tuhhosg/reupnix*. https://doi.org/10.5281/zenodo.7929610

[19] Matthew Kenigsberg. 2021. *A Performance and Storage Evaluation of Lightweight Containerization with NixOS.* Master's thesis. Vanderbilt University. https://ir.vanderbilt.edu/bitstream/handle/1803/16648/KENIGSBERG-THESIS-2021.pdf

[20] Markus Kowalewski and Phillip Seeber. 2022. Sustainable packaging of quantum chemistry software with the Nix package manager. *International Journal of Quantum Chemistry* 122, 9 (2022), e26872. https://doi.org/10.1002/qua.26872

[21] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Network and Distributed Systems Security Symposium.* The Internet Society. https://www.ndss-symposium.org/ndss2013/attack-surface-metrics-and-automated-compile-time-os-kernel-tailoring

[22] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (2022), 62–70. https://doi.org/10.1109/MS.2021.3073045

[23] Hannu Leppinen. 2017. Current Use of Linux in Spacecraft Flight Software. *IEEE Aerospace and Electronic Systems Magazine* 32, 10 (2017), 4–13. https://doi.org/10.1109/MAES.2017.160182

[24] Linux Foundation. [n. d.]. *Yocto Project.* https://www.yoctoproject.org

[25] Linux Foundation. 2021. *OCI Image Format Specification v1.0.* https://github.com/opencontainers/image-spec/blob/main/spec.md

[26] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 239, 2 (2014), 2. https://doi.org/10.5555/2600239.2600241

[27] Josefine Nittler and Mattias Ahlsén. 2021. Key components of building customer trust in the space industry: An investigation of the future of satellite applications.

[28] Colin Percival. 2003. Naive Differences of Executable Code. *Draft Paper, http://www.daemonology.net/bsdiff* (2003). http://www.daemonology.net/papers/bsdiff.pdf

[29] Colin Percival. 2006. *Matching with mismatches and assorted applications.* Ph. D. Dissertation. University of Oxford. https://ora.ox.ac.uk/objects/uuid:4f0d53cc-fb9f-4246-a835-3c8734eba735

[30] Lennart Poettering et al. 2010. *System and Service Manager.* https://systemd.io/

[31] J. Praks, M. Rizwan Mughal, R. Vainio, P. Janhunen, J. Envall, P. Oleynik, A. Näsilä, H. Leppinen, P. Niemelä, A. Slavinskis, J. Gieseler, P. Toivanen, T. Tikka, T. Peltola, A. Bosser, G. Schwarzkopf, N. Jovanovic, B. Riwanto, A. Kestilä, A. Punkkinen, R. Punkkinen, H.-P. Hedman, T. Säntti, J.-O. Lill, J.M.K. Slotte, H. Kettunen, and A. Virtanen. 2021. Aalto-1, multi-payload CubeSat: Design, integration and launch. *Acta Astronautica* 187 (2021), 370–383. https://doi.org/10.1016/j.actaastro.2020.11.042

[32] Hongwei Qin, Dan Feng, Wei Tong, Yutong Zhao, Sheng Qiu, Fei Liu, and Shu Li. 2021. Better atomic writes by exposing the flash out-of-band area to file systems. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems.* 12–23. https://doi.org/10.1145/3461648.3463843

[33] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. 2014. Automatic Feature Selection in Large-Scale System-Software Product Lines. In *Proceedings of the 13th International Conference on Generative Programming and Component Engineering (GPCE '14)*, Matthew Flatt (Ed.). ACM Press, New York, NY, USA, 39–48. https://doi.org/10.1145/2658761.2658767

[34] Rusty Russell, Daniel Quinlan, Christopher Yeoh, and Contributors. 2015. *Filesystem Hierarchy Standard, Version 3.0.* Technical Report. The Linux Foundation. https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf

[35] Chris Simmonds. [n. d.]. *mender.io – Open source over-the-air software updates for Linux devices.* https://mender.io/

[36] IBM Space. 2021. *A Ground Breaking Cubesat Mission: ENDURANCE.* https://endurancein.space/

[37] Christian Stewart. 2021. SkiffOS: Minimal Cross-compiled Linux for Embedded Containers. https://doi.org/10.48550/arXiv.2104.00048

[38] Chris Stirrat. 2020. *The Evolution of Android OTA: A/B Updates.* esper. https://blog.esper.io/the-evolution-of-android-ota-updates/ Accessed 2022-08-12.

[39] Carl von Platen and Johan Eker. 2006. Feedback Linking: Optimizing Object Code Layout for Updates. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems* (Ottawa, Ontario, Canada) *(LCTES '06)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/1134650.1134653

[40] Li Wang, Shuaijun Liu, Weidong Wang, and Zhiyan Fan. 2020. Dynamic Uplink Transmission Scheduling for Satellite Internet of Things Applications. *China Communications* 17, 10 (2020), 241–248. https://doi.org/10.23919/JCC.2020.10.018

Received 2023-03-16; accepted 2023-04-21