EROFS

Release 0.1

EROFS filesystem developers

Sep 02, 2026

GET STARTED

EROFS - Enhanced Read-Only File System

A modern, flexible, general-purpose, high-performance, block-based immutable filesystem with a highly optimized on-disk format and runtime implementation, designed for a variety of use cases—not just storage space savings—while maintaining optimal runtime performance.

EROFS has been formally available since Linux 5.4. It is currently maintained by an open-source community from all over the world, and is still *under active development*.

Get Started

FAQ

Advanced designs A modern filesystem more than just another archive format: EROFS is *strictly* block-aligned to maximize the data utilization of a single disk I/O and enable advanced features like Direct I/O and FSDAX.

Learn more »

Technical Design Highly configurable Minimal core on-disk format for non-encoded use cases. Besides, advanced on-disk and/or runtime features can be enabled on demand.

Learn more »

Features and Comparison Compression & Deduplication Per-file data compression as an option, normally using fixed-sized output compression to fill up each block. Compressed data can be deduplicated with byte-granularity cut points.

Learn more »

Block-aligned fitblk compression

Applications

- The Android system firmware
- Composefs; containerd; Dragonfly Nydus; gVisor; Kata Containers; Kata Containers (rootfs); nerdbox
- Archiso; CoreOS Assembler; dracut-ng; kdump-utils; KIWI; Linglong
- Agent Sandboxes: Super Rad Company microsandbox; Tencent Cloud Cube Sandbox [intro (chn)]; openEuler Conch [intro (chn)]
- Distributions: Alpine Linux; Amazon Linux; Arch Linux; Azure Linux; Bottlerocket; Buildroot; CentOS Stream; Chromium OS; Container-Optimized OS; Debian; Deepin Linux; Fedora; Gentoo; GNU Guix; Homebrew; MacPorts; NixOS; openAnolis; OpenCloudOS; openEuler; openSUSE; OpenWrt; Oracle Linux; Ubuntu; Red Hat Enterprise Linux; Void Linux; yocto; and maybe more.
- Bootloaders: Das U-Boot; GNU GRUB

Architecture

Feedback & Contributing

If you're interested, feel free to send feedback and/or patches to the linux-erofs mailing list <linux-erofs@lists.ozlabs.org> or join our Matrix room <#erofs:matrix.org>. Developer guides might be a useful start for newcomers as the first step.

Please also take a look at Code of Conduct for all Linux kernel development communities.

Additional resources

EROFS in-tree documentation - kernel.org

An introduction to EROFS - LWN.net

EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices - USENIX ATC'19.

EROFS (chn) - Weixin Official Accounts Platform

EROFS Everywhere: An Image-Based Kernel Approach for Various Use Cases (chn) with slides (eng) - KubeCon + CloudNativeCon + Open Source Summit China 2023

Finding the Best Block Filesystem for Your Embedded Linux System - Michael Opdenacker, Bootlin @ EOSS 2023

CHAPTER

ONE

INSTALLATION

1.1 Install erofs-utils via Package Managers

erofs-utils is available on most popular Linux distributions, although it may not be the latest stable version.

Here are examples for `Arch Linux`, `Debian`, `Fedora`, `OpenAnolis` and `Ubuntu`:

```
# Arch Linux
$ sudo pacman -S erofs-utils

# Debian and Ubuntu
$ sudo apt install -y erofs-utils

# Fedora and OpenAnolis
$ sudo dnf install -y erofs-utils
```

On macOS, erofs-utils can be installed using `MacPorts` or `Homebrew`:

```
# MacPorts
$ sudo port install erofs-utils

# Homebrew
$ brew install erofs-utils
```

1.2 Build from source

1.2.1 Install build dependencies

To build erofs-utils, the following dependencies will be needed:

```
# Debian and Ubuntu
$ sudo apt install -y autoconf automake libfuse-dev liblz4-dev liblzma-dev libtool␣
↪libzstd-dev pkg-config uuid-dev zlib1g-dev

# Fedora and OpenAnolis
$ sudo dnf install -y autoconf automake fuse-devel libtool libuuid-devel libzstd-devel␣
↪lz4-devel pkg-config xz-devel zlib-devel
```

1.2.2 Download the erofs-utils source code

Use Git to clone the erofs-utils repository:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/xiang/erofs-utils.git
```

The repo defaults to the master development branch. You can also check out a release tag to build:

```
$ git checkout <tag> # v1.9.1, v1.8.10, v1.7.1, etc.
```

1.2.3 Configure and build

Please run `./autogen.sh; ./configure` from the repository's root directory. `./configure` will prompt you for the usability of erofs-utils dependencies and asks for additional build configuration options:

```
$ ./autogen.sh
$ ./configure --enable-lz4 --enable-lzma --enable-fuse --with-libzstd
$ make
```

1.2.4 Install erofs-utils

Use `make install` to install the generated files of erofs-utils:

```
$ sudo make install
```

CHAPTER

TWO

BUILD & MOUNT

2.1 Creating an EROFS filesystem

To create an EROFS image, use the **mkfs.erofs(1)** utility provided by erofs-utils.

2.1.1 From a local directory

You can create an EROFS image directly from a local directory:

```
$ mkfs.erofs [OPTIONS] <IMAGE> <SOURCE DIRECTORY>
```

By default, `mkfs.erofs` creates plain images, as compression is an optional on-disk feature. Compression and other behaviors can be configured via command-line options.

Here are some frequently used options for mkfs.erofs:

Option	Description
-z X[,level=#]	Specify a valid compressor[1] and, optionally, a compression level.
-C #	Specify the physical cluster size for compression (default: filesystem block size).
-b #	Specify the filesystem block size (default: system page size, e.g. 4096).
-T #	Set the UNIX timestamp for the filesystem. It will behave as `--all-time` if `--mkfs-time` is not specified.
–all-time	(used together with `-T`) Set all files to the fixed timestamp.
–mkfs-time	(used together with `-T`) The given timestamp is only applied to the image creation time.
-U X	Specify the filesystem UUID, or one of the following value: clear, random.
-E ztailpacking	Inline the tail parts of compressed files into their metadata.
-E fragments	Pack the tail parts of compressed files, or entire files if they are small, into a special inode.
-E all-fragments	(not recommended) Pack entire files into a special inode to reduce image size.
-E dedupe	Enable global data deduplication[2]. Note: not supported with multi-threading *yet*.

For example, to generate an uncompressed EROFS image from `foo/`:

```
$ mkfs.erofs foo.erofs foo/
```

[1] Supported compressors: lz4, lz4hc, lzma, deflate, libdeflate, and zstd.

[2] If data compression (encoded data) is enabled, rolling-hash-based deduplication will be used; otherwise, block-based deduplication will be applied.

To generate a compressed EROFS image using LZ4HC (level 12), with `fragments` and `ztailpacking` features enabled, and the physical cluster size of 65536:

```
$ mkfs.erofs -zlz4hc,12 -C65536 -Efragments,ztailpacking foo.erofs foo/
```

Note: By default, EROFS just uses a physical cluster size equal to the block size (e.g., 4096 on x86) and disables advanced features to keep the random access performance. Consider increasing `-C` (e.g., to 65536 or 131072) and enabling advanced features if a smaller image size is desired. Note that larger cluster sizes may increase memory footprint during decompression and degrade random access performance.

Using `-Eall-fragments` is not recommended now, as it can degrade runtime performance unless minimizing image size is the top priority. `-Efragments` already supports multi-threading.

2.1.2 From a tarball file

Alternatively, an EROFS image can also be generated from a tarball file using the `--tar` command-line option.

Here are some frequently used additional options for `mkfs.erofs` tar mode:

Option	Description
–tar=f	Generate a full EROFS image from a regular tarball.
–tar=i	Generate a meta-only EROFS image from a regular tarball.
–aufs	Convert AUFS special files to OverlayFS metadata (commonly used in Docker/OCI images).
–sort=none	(only valid with `--tar=f`) Keep inode data order consistent with tar data order.
–sort=path	(only valid with `--tar=f`) Data order strictly follows the tree generation order (default).

For example, to generate a full EROFS image from a tarball `foo.tar`:

```
$ mkfs.erofs --tar=f --sort=none foo.erofs foo.tar
```

Note that `--sort=none` is used if strict data order is not required; it helps eliminate unnecessary data writes.

Additionally, `--tar=i` can be used to generate a minimized EROFS metadata index that references external tar data:

```
$ mkfs.erofs --tar=i foo.erofs foo.tar
```

The generated tar index EROFS image can be used with the original tar file specified using the mount option `-odevice=`. Alternatively, you can append the original tar file to the tar index:

```
cat foo.tar >> foo.erofs
```

2.2 Mount

To mount an EROFS image, use the **mount(8)** command on a block device or a regular EROFS image file, as shown below:

```
$ mkdir ~/mnt
$ sudo mount /dev/sdX ~/mnt
```

All files contained in `/dev/sdX` will now be accessible under the `~/mnt` mount point.

If the image is a regular file and the command above doesn't work (e.g., due to an older version of util-linux), you can use the `-o loop` option:

```
$ mkdir ~/mnt
$ sudo mount -o loop foo.erofs ~/mnt
```

Alternatively, unprivileged users can mount an EROFS image using **erofsfuse**:

```
$ mkdir ~/mnt
$ erofsfuse foo.erofs ~/mnt
```

2.3 Unmount

To unmount the filesystem, use the **umount (8)** command for privileged users:

```
$ sudo umount ~/mnt
```

For unprivileged users, you could also use the **fusermount** command to unmount an instance out of `erofsfuse`:

```
$ fusermount -u ~/mnt
```

CHAPTER

THREE

FEATURES AND COMPARISON

3.1 Image filesystem

Container technologies like cgroups and namespaces offer strong isolation and portability. When applications are manually maintained, they can still end up with outdated dependencies or inconsistent configurations. This inconsistency demands constant monitoring and manual intervention, which quickly becomes unmanageable at scale.

Therefore, the industry consensus for addressing this is to introduce immutable infrastructure: just like the famous Docker slogan "Build once, run anywhere", immutable images make every container consistent with each other, providing strong reliability and reducing deployment failures. In addition, since every new deployment starts from a clean, verified state, it's much easier to audit and scale, which improves security and scalability. That's why container images have become so popular today, in the cloud-native and AI eras.

3.1.1 Layering

Since containers are isolated, lightweight silos for running applications on the host operating system, it makes sense to keep those immutable container images lightweight as well.

One way to achieve this is through deduplication across images. Today, Docker/OCI container images are composed of layers to support incremental builds, enabling files in shared layers to be reused across different images.

The key technology enabling layering is called `snapshotting`. To share layers among various physical nodes, snapshots should be distributable easily too. Currently, there are three main approaches to snapshotting:

1. The classic VM disk image approach;
2. Using an existing copy-on-write filesystem that supports snapshots, such as BTRFS or ZFS;
3. Using a dedicated archive format or an effective image filesystem.

The following subsection explores these approaches in more detail.

3.1.2 Why not VM disk images or CoW filesystems?

You may expect it is all about the image filesystem as the title suggests, but why do we use an image filesystem instead of distributing VM disk images or simply using mature filesystem snapshotting to manage container images?

First of all, an answer like "tar.gz is the de facto standard and it uses the tar package format, so we'd better use a filesystem to overcome all the shortcomings of the tar format" is not helpful because it relies on historical precedent and does not address any key point.

Here are two main reasons:

Containerization requires a precise, auditable format

Indeed, the classic VM disk image approaches can also be used for container images, and disk snapshot formats like Qcow2 can be used in order to support layering. They do address the main issues of tar because they add another level of abstraction for layering support and expect existing black-box kernel filesystems to resolve the remaining pain points. However, the main concern with those approaches is the lack of precision: any format can be included as long as some component (e.g., the host kernel) can handle it.

Unlike virtualization, which provides very strong isolation, containerization typically shares a single kernel, requiring greater cooperation and trust. As a result, any data from a remote source could become a vulnerability for the entire system. That means container images must be safe for both virtualization and containerization. In other words, every user must know exactly what is inside the container image, making the classic VM disk image approaches insecure if an unknown filesystem format is mounted by the host kernel and a precise white-box format definition essential for auditing and scanning to find security vulnerabilities.

Generic filesystems pose consistency and stability risks

Generic filesystems can serve as precise, white-box formats for container images. However, they are still vulnerable when used as remote images due to immitigable metadata inconsistencies.

For example, the allocation status of a physical block may be recorded in multiple places: the allocation tree (or bitmap), inode extents, and may also appear in a reverse mapping tree. When an image is fetched from an untrusted remote source, an attacker can craft inconsistencies and the resulting bugs are serious and hard to prevent.

Performing extra consistency checks, either at runtime or beforehand with a tool like `fsck(8)`, incurs heavy performance penalties. These risks will then be amplified by the complexity of kernel filesystem implementations, posing additional threats to the hosts.

Other internal metadata, such as filesystem journaling, which can cause extra inconsistency, which is similar, so no need to go into further detail here.

Note: For example, the following crafted EXT4 image can immediately crash all Linux kernel versions, since it uses an expected by-design behavior in EXT4:

It does not cause any obscure metadata inconsistency; it just corrupts the root inode and sets `s_errors(EXT4_ERRORS_PANIC)` using the following `debugfs` commands:

```
debugfs: set_super_value errors 3
debugfs: sif <2> bmap[0] 0
```

In addition, there are known EXT4 syzkaller bugs that could be exploited (51 open bugs as of 12/03/2025), making mounting untrusted remote EXT4 filesystems on the host absolutely unsafe.

Disclaimer: This paragraph is only used to explain technical details. Any further harmful exploits have no relationship with the EROFS project.

3.1.3 The solution

Our way to resolve this is *read/write separation at the filesystem level*: distribute remote data in a reliable, read-only archive format to prevent any serious inconsistencies, and then prepare the writable layer by reusing a trusted generic filesystem or generating a new filesystem locally.

By the way, it does NOT mean a simple archive format won't have any inconsistency or corruption, but since it only contains necessary metadata like inodes and directories, either such inconsistency is not harmful (and can even be ignored) or should be a common issue either resolved in the VFS (e.g. *directory hardlinks*) or needs to be resolved by any specific filesystem which needs to parse filesystem metadata, including FUSE.

EROFS is designed as a simple, flexible immutable filesystem format similar to previous archive formats such as tar, zip, and cpio as well as the CD-ROM filesystems with the following advanced highlights:

- Since it's just like an advanced archive format *(basically equivalent to unpacking packages)*, its metadata cannot become severely inconsistent, and the kernel implementation should be able to bear any on-disk corruption by design;
- Data is strictly aligned on disk for extreme performance, and it enables other possibilities, such as FSDAX and global data deduplication, from the storage stack perspective (e.g., filesystem reflinks and disk deduplication, which are difficult with unaligned archive formats);
- It implements built-in compression, deduplication, and native layering features, which can be helpful for containerization;
- It supports native kernel file-backed mounts, so EROFS images can be mounted on Linux natively without loopback block devices;
- Like common archive formats, EROFS images can be distributed as golden data and stored on any filesystem. Depending on user requirements or specific workloads, the writable layer can also be freely configured with any supported filesystem using OverlayFS instead of purely relying on remote metadata.

Due to its flexibility and simplicity, it is well-suited for use in container images and sandbox templates among runc and VM-based containers, making it easier for end users to audit and scan for vulnerabilities in the images and reduce potential risks.

3.1.4 Use Cases (Alphabetical)

1. Composefs
2. Containerd EROFS snapshotter
3. Dragonfly Nydus

3.2 Deduplication among versions

3.2.1 Minor container image updates

- Examine 10 versions of ubuntu:jammy from Docker Hub, specifically jammy-{20221130,20230126,20230301,20230308,20230425,20230522,20230605,20230624,20230804,20230816};
- These images only have one layer which is in the tar.gz format;
- Uncompressed EROFS images are built with `-Ededupe` and compressed EROFS images are built with `-Ededupe,all-fragments` and LZ4HC, DEFLATE algorithms.

	Total Size (MiB)	Average layer size (MiB)	Saved / 766.1MiB
Compressed OCI (tar.gz)	282.5	28.3	63%
Uncompressed OCI (tar)	766.1	76.6	0%
Uncompressed EROFS	109.5	11.0	86%
EROFS (DEFLATE,9,32k)	46.4	4.6	94%
EROFS (LZ4HC,12,64k)	54.2	5.4	93%
SquashFS (GZIP,9,128k,-noI[1])	47.0	4.7	94%
SquashFS (LZ4HC,12,128k,-noI)	54.7	5.5	93%

It shows that EROFS gets **smaller sizes** even with smaller compression granularity.

3.2.2 Wikipedia snapshots

Two snapshots were selected: `20230201` and `20230220`:

- Each snapshot included the first 100 pages at that time.
- There was 20-day difference between each other.

	Deduped?	Size (MiB)	Saved / 1888MiB
Uncompressed text	No	1888	0%
SquashFS [4k][2]	No	1235	34.6%
EROFS [4k, *default*]	No	1201	36.4%
SquashFS [8k]	No	1150	39.1%
EROFS [4k][3]	Yes	1147	39.2%
SquashFS [128k, *default*]	No	1110	41.2%
EROFS [64k]	Yes	988	47.7%

3.2.3 Linux kernel source code releases

Three kernel releases are selected: `5.10`, `5.10.50`, `5.10.100`:

- Images from both filesystems are compressed with `LZ4HC, 12` since SquashFS uses the maximum level 12 for LZ4HC;
- EROFS images are built with `-T0 --force-uid=1000 --force-gid=100` in order to force 32-byte inodes to approximately match SquashFS metadata size;

[1] SquashFS uses `-b 131072` by default, `-noI` will disable its metadata compression.
[2] SquashFS command line: `-comp lz4 -b # -noappend`
[3] EROFS command line: `-zlz4 -Ededupe -C #`

	Fragments?	Deduped?	Size
SquashFS [16k]	Yes	No	409 MiB
EROFS [4k, *default*]	Yes	Yes	379 MiB
SquashFS [32k]	Yes	No	365 MiB
EROFS [8k]	Yes	Yes	347 MiB
SquashFS [64k]	Yes	No	334 MiB
EROFS [32k]	Yes	Yes	313 MiB
SquashFS [128k, *default*]	Yes	No	312 MiB
EROFS [64k]	Yes	Yes	310 MiB

Note that it's just **an incomplete list** for the qualitative evaluation. The overall purpose of this is to show EROFS benefits compared to other in-kernel approaches when making technical decisions.

Feature (as of Linux 6.17)	EROFS	EXT4	SquashFS
Minimal block size	512 B[1]	1 KiB	Unaligned[2]
Inode size	32/64 B	128/256 B	Varied[3]
Limitation of total UIDs/GIDs	Unlimited	Unlimited	65536[4]
Pre-1970 / ns timestamps	Yes	Yes	No
Filesystem UUID	Yes	Yes	No
Filesystem label (Volume label)	Yes	Yes	No
Inline data	Yes (Inline tail)	Yes	No
Data compression	Yes[5]	No	Yes
Largest compression granularity	1 MiB	N/A	1 MiB
Default compression granularity	1 Block[6]	N/A	128 KiB
Fragments	Yes	N/A	Yes
File-backed mounts	Yes[7]	No	No
Metadata compression	Yes[8]	N/A	Yes
Multiple compression algorithms	Per-file	N/A	No
Data deduplication	Extent-based	No?[9]	File-based
Extended attribute support	Yes	Yes	Yes
External data (multi-devices)	Yes	No	No
POSIX.1e ACL support	Yes	Yes	No
Direct I/O support[10]	Yes	Yes	No
FIEMAP support	Yes	Yes	No
SEEK_{DATA,HOLE} support	Yes	Yes	No
FSDAX support	Yes	Yes	No
Large folio support	Yes[11]	Yes[12]	No
Hardware acceleration support	Yes[13]	No	No

[1] 512-byte blocks can be used for tarball data reference.

[2] It means a fixed minimal filesystem I/O size which SquashFS doesn't have. Instead, SquashFS has its own "block size": its compressed files are split up in fixed-size blocks. See Squashfs Binary Format/About.

[3] SquashFS has different on-disk inodes for each type of varying contents and size. See Squashfs Binary Format/Inode Table.

[4] SquashFS allows 32-bit UIDs/GIDs, but only among 2^{16} unique values. See Squashfs Binary Format/Inode Table.

[5] Data compression is an optional feature of the EROFS filesystem. Currently, the supported compression algorithms include LZ4, MicroLZMA (since Linux 5.16), DEFLATE (since Linux 6.6 LTS) and Zstandard (since Linux 6.10).

[6] The default block size of EROFS is 4 KiB on x86 and x86-64.

[7] EROFS has supported EROFS over fscache (since Linux 5.19, deprecated in Linux 6.12) and file-backed mounts (since Linux 6.12) to avoid unnecessary loop devices.

[8] EROFS metadata is designed to be directly accessible from block devices without decoding, or deserialization, as these can lead to I/O am-

plification and additional runtime latency in performance-critical scenarios. Nevertheless, optional metadata compression has been supported since Linux 6.17 to minimize filesystem images.

[9] Strictly speaking, EXT4 has a feature named "shared_blocks", which will prevent applications from writing to the filesystem.

[10] For example, `direct I/O` can be used for loop devices backed by unencoded files on the EROFS filesystem to avoid double caching. `Direct I/O` on encoded files is almost useless since it should not do ANY caching and thus will kill the overall performance.

[11] EROFS has supported large folios for uncompressed files (since Linux 6.2) and compressed files (since Linux 6.11).

[12] EXT4 supports large folios since Linux 6.16.

[13] EROFS has supported Intel QuickAssist Technology to accelerate DEFLATE algorithm since Linux 6.16.

CHAPTER

FOUR

CASE STUDIES

4.1 Bootable system images

4.1.1 Android system partitions

EROFS was originally developed for this use case in 2017. Nowadays, all mainstream Android vendors use EROFS for their system partitions, leveraging its high-performance compression to reduce image sizes without any noticeable impact on user experience.

Known adopters include (alphabetical, may be incomplete):

- Coolpad
- Honor
- Huawei
- Motorola
- Samsung
- OPPO
- Xiaomi
- vivo

For more details on Android system partition use cases, see:

- EROFS AOSP user guide
- ATC'19 EROFS

4.1.2 Fedora LiveCDs

Since Fedora 42, the EROFS filesystem has been used to compress Fedora LiveCDs to achieve smaller images with even better runtime performance.

For more details, see Switch to EROFS for Live Media.

The EROFS filesystem has been deployed in production at large scale since its inclusion in Linux 5.4 and is now part of several prominent open-source ecosystems.

It started in 2017, when Linux image filesystems were still primitive (cramfs and ROMFS), or focused solely on extreme image size reduction (squashfs) for low-end embedded devices, while lacking good real-time response in low

memory scenarios. In addition, kernel development for most in-tree immutable filesystems (particularly on-disk format enhancements) was inactive for a decade at that time.

To meet our production performance needs and fill the gap for a better high-performance Linux image filesystem, the EROFS project was launched. Before being successfully upstreamed, it had already been successfully deployed on millions of Android smartphones and it should be billions of various devices now.

Note that the EROFS filesystem aims to be a generic image filesystem from the beginning, not limited to a specific use case (e.g. only for Android system partitions). We believe that even if image filesystems have certain useful use cases, in reality it often fails to attract many real experienced kernel filesystem developers. Designing a specialized filesystem for a single use case will lead to unhealthy development, unnecessary reinvention, and fragmentation.

Currently, apart from the initial Android use cases, we are also focused on bringing up a better image filesystem for containers. Many recent features, such as native layering support, large folios, file-backed mounts, inode-based page cache sharing, and direct access for files, are dedicated to container image use cases.

In several popular projects, EROFS is used as a powerful tool to provide advanced features or improve the de-facto standard Docker/OCI container images, such as ComposeFS and the EROFS containerd snapshotter. However, EROFS can also serve as a next-generation container filesystem on its own. We believe it should be more useful as the EROFS container ecosystem continues to evolve.

The EROFS filesystem is now a completely community-driven, neutral open-source project and part of the Linux kernel. If you're working on, or interested in image filesystem use cases, feel free to provide feedback and join us.

Detailed introduction to the current main use cases is still in progress.

CHAPTER

FIVE

TECHNICAL DESIGN

The following sections are a summarized technical description for reference if you are a user or developer interested in EROFS internals. There is also a paper available, EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices. The details defined in the paper are slightly outdated but the overall ideas are nearly the same.

5.1 Block-aligned vs unaligned

EROFS data is all arranged in **fixed-size blocks** (aka. block-aligned, typically 4 KiB) like many modern disk filesystems (e.g. ext4, xfs, btrfs, f2fs, etc.) to match the intrinsic characteristics of block devices. This means the corresponding data can be *directly* parsed by reading a single filesystem block for **non-encoded data** or (possibly) multiple consecutive blocks (called a single physical cluster) for **encoded data**, which differs from *archive formats or unaligned filesystems* (e.g. `cramfs`, `romfs`, `squashfs`, `affs`, and possibly more FUSE-based implementations.)

The main benefits of using fixed-size blocks are:

- Block-aligned non-encoded EROFS data can be directly loaded into kernel page cache and memory-mapped into user space **without any extra post-processing**. Direct I/O and FSDAX can also be used for block-aligned non-encoded EROFS data.
- Block-aligned encoded EROFS data I/O can be **fully utilized**, which means there is no unusable encoded byte in each single I/O request. Unlike other unaligned approaches, EROFS does not have to cache such unusable encoded bytes for later decoding (or never used) for better performance (otherwise I/O efficiency for small random I/Os is low due to undecodable encoded bytes), which is considered harmful due to utilizing a larger memory footprint than necessary.
- Alternatively, if the encoded blocks are not utilized immediately (esp. very little decoded data is requested), the encoded blocks could still be *cached in the page cache for later use*. This is quite useful on memory-limited devices since *caching compressed data is generally more efficient than decompressed data if selected compression algorithms are **fast** enough* (considering main benefits of ZSWAP or ZRAM). Although unaligned solutions could also cache encoded data in page cache, reclaiming could lead to lower cache pages utilization due to fragmentation since page cache reclaims data in pages instead of bytes.
- Because of this, block-aligned encoded EROFS makes **compressed data cached independently**, which means a single physical cluster could be cached without coupling with other compression units. In other words, unlike unaligned solutions, it is more flexible for EROFS to only cache the necessary physical clusters.

5.2 Block-aligned fitblk compression

In addition to block-aligned data, unlike EXT4, XFS, BTRFS, F2FS, etc., EROFS mainly uses a **fixed-size output compression** approach to increase encoded block utilization and maximize compression ratios.

Such compression approach is not required. However, without this approach, the final blocks of physical clusters will not be fully filled with encoded data. Currently LZ4, LZMA (Linux 5.16+), DEFLATE (Linux 6.6+), and Zstandard (Linux 6.10+) algorithms natively support this mode.

Fixed-size output compression generally provides better compression ratios, especially on small physical clusters (e.g., 4 or 8 KiB), saving about 5% additional space.

Note that **small compressed physical clusters** are quite important to end-to-end performance for memory-intensive workloads (that is exactly the EROFS main target case) since it's quite hard to fully cache either (de)compressed data in memory on such extreme workloads. In other words, reloading can happen frequently due to cache misses on these workloads.

5.3 Data deduplication

EROFS supports both *fixed-size chunk deduplication* and *compressed data deduplication*:

- For non-encoded files, data can be split into fixed-size chunks for mmap-I/O friendly and only keep chunks with different contents on disk.
- For encoded files, each physical cluster can be used in a similar way for multiple reference. Note that cut points will be adjusted using the Rabin–Karp algorithm in order to improve deduplication (decrease likelihood of unique blocks).

5.4 In short: how is EROFS designed for performance?

A summary of our overall design is listed below:

- Block-aligned data - no additional I/O waste;
- Fixed-size output compression - better compression ratios and efficient I/O utilization;
- Independent cache or in-place I/O strategies - to minimize memory footprints;
- In-place decompression - avoid bounced compressed buffers and poisoned cache lines as much as possible;
- Multi-reference compressed clusters - avoid deduplicated data I/O (where possible) and minimize images even further.

Note: EROFS does not prefetch unnecessary on-disk data or amplify user I/O at runtime, unlike some alternative approaches, which degrades random I/O performance and increases memory footprint. Those customized strategies can also be achieved by using posix_fadvise() or user space tools like `ureadahead`, so there is no need to bother the kernel.

5.4.1 EROFS On-disk Format

EROFS uses a flexible, hierarchical, block-aligned on-disk layout that is built with the following goals:

- DMA- and mmap-friendly, block-aligned data to maximize runtime performance on all kinds of storage devices;
- A simple core on-disk format that is easy to parse and has zero unnecessary metadata redundancy for archive use unlike other generic filesystems, ideal for data auditing and accessing remote untrusted data;
- Advanced on-disk features like compression (compressed inodes and metadata compression) are completely optional and aren't mixed with the core design: you can use them only when needed.

The entire filesystem tree is built from just three core on-disk structures:

- **Superblock** — located at a fixed offset of 1024 bytes; the only structure at a fixed position in the filesystem.
- **Compact/Extended inodes** — per regular file, device, symlink, or directory; addressed in O(1) time via a simple NID-to-offset formula.
- **Directory entries** — 12-byte records, sorted lexicographically by filename at the beginning of each directory block (each data block of a directory inode).

Optional features extend this foundation without breaking the core design:

- ***Extended attributes (xattrs)*** support per-inode metadata. Several mechanisms — including a shared xattr pool, long prefix tables, and a per-inode Bloom filter — keep storage overhead low even when xattrs are used extensively.
- ***Chunk-based layout*** splits large files into fixed-size, independently-addressed chunks, enabling cross-file deduplication and multi-device storage.

Core On-disk Format

Overview

The EROFS core on-disk format is designed to be **as simple as possible**, since one of the basic use cases of EROFS is as a drop-in replacement for tar or cpio:

The format design principles are as follows:

- Data (except for *inline data*) is always block-based; metadata is not strictly block-based.
- There are **no centralized inode or directory tables**. These are not suitable for image incremental updates, metadata flexibility, and extensibility. It is up to users to determine whether inodes or directories are arranged one by one or not.
- I/O amplification from **extra metadata access** should be as small as possible.

There are *only **three** on-disk components to form a full filesystem tree*: superblock, inodes, and directory entries.

Note that only the superblock needs to be kept at a fixed offset, as mentioned below.

Conformance to Core Format

An EROFS image conforms to the core on-disk format if and only if **all** of the following conditions are met:

1. The `is_compressed` field (offset 0x54, 2 bytes) in the superblock is **0**.
2. All bits in `feature_compat` and `feature_incompat`, except those listed in the *Feature Flags* section below, are **0**.

An image that does not meet these conditions uses one or more optional features described in separate feature-specific documents. Note that the core on-disk format has always been supported since Linux 5.4; thus, the 48-bit layout is not part of the core on-disk format (for example), and not all users need 48-bit block addressing.

Superblock

The EROFS superblock is located at a fixed absolute offset of **1024 bytes**. Its base size is 128 bytes. When `sb_extslots` is non-zero, the total superblock size is `128 + sb_extslots * 16` bytes. The first 1024 bytes are unused, which allows for support of other advanced formats based on EROFS, as well as the installation of x86 boot sectors and other oddities.

Field Definitions

Offset	Size	Type	Name	Description
0x00	4	`u32`	`magic`	Magic signature: `0xE0F5E1E2`
0x04	4	`u32`	`checksum`	CRC32-C checksum of the superblock block; see *Superblock Checksum*
0x08	4	`u32`	`feature_compa`	Compatible feature flags; see *Feature Flags*
0x0C	1	`u8`	`blkszbits`	Block size = `2^blkszbits`; minimum 9
0x0D	1	`u8`	`sb_extslots`	Number of 16-byte superblock extension slots
0x0E	2	`u16`	`rootnid_2b`	Root directory NID
0x10	8	`u64`	`inos`	Total inode count; see *blocks and inos Fields*
0x18	8	`u64`	`epoch`	Filesystem creation time, seconds since UNIX epoch
0x20	4	`u32`	`fixed_nsec`	Nanoseconds component shared by all compact inodes; see *Modification Time in Compact Inodes*
0x24	4	`u32`	`blocks`	Total block count; see *blocks and inos Fields*
0x28	4	`u32`	`meta_blkaddr`	Start block address to specify the inode-metadata zone
0x2C	4	`u32`	`xattr_blkaddr`	Start block address to specify the extended attribute zone
0x30	16	`u8[]`	`uuid`	128-bit UUID for the volume
0x40	16	`u8[]`	`volume_name`	Filesystem label (not null-terminated if 16 bytes)
0x50	4	`u32`	`feature_incom`	Incompatible feature flags; see *Feature Flags*
0x54	2	`u16`	`is_compressed`	0 for non-compressed images, any non-zero value for compressed images
0x56	4	`u32`	*dontcare*	External device support specific; ignored in core format
0x5A	1	`u8`	`dirblkbits`	Directory block size = `2^(blkszbits + dirblkbits)`; strictly 0 in the core format
0x5B	5	`u8[]`	*dontcare*	Xattr specific; ignored in core format
0x60	8	`u8[]`	*dontcare*	Compression specific; ignored in core format
0x68	1	`u8`	*reserved*	Reserved; must be 0
0x69	1	`u8`	*dontcare*	Xattr specific; ignored in core format
0x6A	2	`u16`	*reserved*	Reserved; must be 0
0x6C	12	`u8[]`	*dontcare*	48-bit layout specific; ignored in core format
0x78	8	`u64`	*reserved*	Reserved; must be 0

Note the difference between *reserved* and *dontcare* fields:

- ***reserved***: Users must not use these fields, and they must be filled with `0` to comply with the supported features or reserve for future use.
- ***dontcare***: Users can safely use these for other purposes as long as the corresponding incompatible feature flag is not set.

Magic Number

The magic number at offset 0x00 must be `0xE0F5E1E2` (little-endian). A reader must reject any image whose first four bytes at offset 1024 do not match this value.

Superblock Checksum

When `EROFS_FEATURE_COMPAT_SB_CHKSUM` is set, the `checksum` field contains a CRC32-C digest. The digest is computed over the byte range `[1024, 1024 + block_size)`, with the four bytes of the `checksum` field itself treated as zero during computation.

For example, when `blkszbits` is 12 (block size is 4 KiB):

Offset	Size	Description	Checksum covered
0	1024	Padding	No
1024	4	Magic number	Yes
1028	4	Checksum field in superblock, filled with zero	Yes
1032	3064	Remaining bytes in the filesystem block	Yes

Tip: Some implementations (e.g., `java.util.zip.CRC32C`) apply a final bit-wise inversion. If the superblock checksum does not match, try inverting it.

Feature Flags

`feature_compat` — Compatible Feature Flags

A mount implementation that does not recognise a bit in `feature_compat` may still mount the filesystem without loss of correctness.

Bit mask	Name	Description
`0x00000001`	`EROFS_FEATURE_COMPAT_SB_CHKS`	Superblock CRC32-C checksum is present; see *Superblock Checksum*
`0x00000002`	`EROFS_FEATURE_COMPAT_MTIME`	Per-inode mtime is stored in extended inodes

Note: For new filesystem builders, it is recommended to always set `EROFS_FEATURE_COMPAT_MTIME`, since it indicates that all inode timestamps record modification time (mtime) rather than change time (ctime).

`feature_incompat` — Incompatible Feature Flags

A runtime implementation that doesn't implement any feature implied by a bit in `feature_incompat` must refuse to mount the entire filesystem.

The core on-disk format defines no incompatible feature flags. A non-zero `feature_incompat` value indicates one or more non-core feature extensions.

`blocks` and `inos` Fields

The `blocks` and `inos` fields are primarily intended for `statvfs(3)` reporting. For dynamically generated EROFS filesystems, these fields can be set to `0`.

Implementations should **not** use the `blocks` field to validate whether a block address or NID is valid. Such checks are unnecessary; malicious block addresses or NIDs will simply result in -EIO or reading corrupted (meta)data without causing any real harmful behaviors. Furthermore, a maliciously crafted image can easily bypass bounds checking by modifying the `blocks` field accordingly, making such validation meaningless.

Inodes

Each on-disk inode must be aligned to a **32-byte inode slot** boundary, which is set to be kept in line with the compact inode size. Given a NID `nid`, its inode can be located in O(1) time by computing the absolute byte offset as follows:

```
inode_offset = meta_blkaddr * block_size + 32 * nid
```

The NIDs for the root directory and special-purpose inodes are stored in the superblock. Valid inode sizes are either **32 bytes** (compact) or **64 bytes** (extended), distinguished by bit 0 of the `i_format` field.

Compact Inode (32 bytes)

Defined as `struct erofs_inode_compact`:

Offset	Size	Type	Name	Description
0x00	2	`u16`	`i_format`	Inode format hints; see *i_format Field*
0x02	2	`u16`	*reserved*	Xattr specific; must be 0 if no xattrs
0x04	2	`u16`	`i_mode`	File type and permission bits
0x06	2	`u16`	`i_nlink`	Hard link count
0x08	4	`u32`	`i_size`	File size in bytes (32-bit)
0x0C	4	`u32`	*reserved*	48-bit layout specific; ignored in core format
0x10	4	`u32`	`i_u`	Union; see *i_u Union*
0x14	4	`u32`	`i_ino`	Inode serial number for 32-bit `stat(2)` compatibility
0x18	2	`u16`	`i_uid`	Owner UID (16-bit)
0x1A	2	`u16`	`i_gid`	Owner GID (16-bit)
0x1C	4	`u32`	*reserved*	Reserved; must be 0

Modification Time in Compact Inodes

Due to space constraints, compact inodes cannot store a full 64-bit per-inode timestamp, let alone an additional nanosecond field. Consequently, when the 48-bit layout extension is unused, the effective timestamp for all compact inodes is `(epoch, fixed_nsec)`, which has been the case since Linux 5.4.

Extended Inode (64 bytes)

Defined as `struct erofs_inode_extended`:

Offset	Size	Type	Name	Description
0x00	2	`u16`	`i_format`	Inode format hints; see *i_format Field*
0x02	2	`u16`	*reserved*	Xattr specific; must be 0 if no xattrs
0x04	2	`u16`	`i_mode`	File type and permission bits
0x06	2	`u16`	*reserved*	Reserved; must be 0
0x08	8	`u64`	`i_size`	File size in bytes (64-bit)
0x10	4	`u32`	`i_u`	Union; see *i_u Union*
0x14	4	`u32`	`i_ino`	Inode serial number for 32-bit `stat(2)` compatibility
0x18	4	`u32`	`i_uid`	Owner UID (32-bit)
0x1C	4	`u32`	`i_gid`	Owner GID (32-bit)
0x20	8	`u64`	`i_mtime`	Modification time, seconds since UNIX epoch
0x28	4	`u32`	`i_mtime_nsec`	Nanoseconds component of `i_mtime`
0x2C	4	`u32`	`i_nlink`	Hard link count (32-bit)
0x30	16	`u8[]`	*reserved*	Reserved; must be 0

`i_format` Field

The `i_format` field is present at offset 0x00 in both inode variants and encodes layout metadata:

Bits	Width	Description
0	1	Inode version: 0 = compact (32-byte), 1 = extended (64-byte)
1–3	3	Data layout: values 0–4 are defined; 5–7 are reserved. See *Inode Data Layouts*
4	1	48-bit layout specific; ignored in core format
5–15	11	Reserved; must be 0

Note: When bits 1–3 contain reserved values (5–7), the inode uses an unsupported data layout. Implementations must reject such inodes and return an appropriate error (e.g., "not supported"). This typically indicates a maliciously crafted or corrupted image.

`i_u` Union

The `i_u` field (4 bytes at offset 0x10) is interpreted based on the data layout:

Name	Applicable when	Description
`i_u.startblk`	Flat inodes	Starting block number
`i_u.rdev`	Character/block device inodes	Device ID

Inode Data Layouts

The data layout of an inode is encoded in bits 1–3 of `i_format`. The core format defines two flat layouts.

`EROFS_INODE_FLAT_PLAIN` (0)

`i_u` is interpreted as `startblk` (the 32-bit starting block address).

The inode's data lies in consecutive blocks starting from that address, occupying `ceil(i_size / block_size)` consecutive blocks.

`EROFS_INODE_FLAT_INLINE` (2)

`i_u` is interpreted as `startblk` (the 32-bit starting block address).

The inode's data lies in consecutive blocks starting from that address, except for the tail part (`i_size % block_size`) that is inlined in the block immediately following the inode metadata. If `i_size` is small enough that the entire content fits in the inline tail, there are no preceding blocks and `i_u` is a don't-care field.

Note: This layout is not allowed if the tail inode data block cannot be inlined (e.g., if inlining the tail data would cause the inode to cross a physical block boundary).

Directories

All on-disk directories are organized in the form of **directory blocks** of size `2^(blkszbits + dirblkbits)`. `dirblkbits` is strictly 0 for now.

Directory Block Structure

Each directory block is divided into two contiguous regions:

1. An array of fixed-size directory entry records starting from the beginning of the block.
2. Variable-length filename strings following the directory entry array.

The `nameoff` field of the **first** entry in a block indicates the total number of directory entries in that block:

```
entry_count = nameoff[0] / sizeof(erofs_dirent)
```

All entries within a directory block, including `.` and `..`, are stored in strict **lexicographic (byte-value ascending) order** to enable an improved prefix binary search algorithm.

Directory Entry Record

Defined as `struct erofs_dirent`:

Offset	Size	Type	Name	Description
0x00	8	`u64`	`nid`	Node number of the target inode
0x08	2	`u16`	`nameoff`	Byte offset of the filename within this directory block
0x0A	1	`u8`	`file_type`	File type code (see below)
0x0B	1	`u8`	*reserved*	Reserved; must be 0

`file_type` Values

Value	Constant	POSIX type
0	`EROFS_FT_UNKNOWN`	Unknown
1	`EROFS_FT_REG_FILE`	Regular file
2	`EROFS_FT_DIR`	Directory
3	`EROFS_FT_CHRDEV`	Character device
4	`EROFS_FT_BLKDEV`	Block device
5	`EROFS_FT_FIFO`	FIFO
6	`EROFS_FT_SOCK`	Socket
7	`EROFS_FT_SYMLINK`	Symbolic link

Filename Encoding

Filenames are not null-terminated (`\0`) except for the last one in each directory block. For each directory block, if the last filename doesn't reach up to the end of the block, the remaining bytes must start with `0x00`.

So the length of entry `i` is derived as:

- For all entries except the last: `nameoff[i+1]  nameoff[i]`.
- For the last entry in the block: `strnlen(filename, block_end  nameoff[last])`.

No character encoding is mandated; UTF-8 is recommended.

Note: Other alternative forms (e.g., `Eytzinger order`) were also considered (that is why there was once .*_classic naming). Here are some reasons those forms were not supported:

- Filenames are variable-sized strings, which makes `Eytzinger order` harder to utilize unless `namehash` is also introduced, but that complicates the overall implementation and expands directory sizes.
- It is harder to keep filenames and directory entries in the same directory block (especially *large directories*) to minimize I/O amplification.
- `readdir(3)` would be impacted too if strict alphabetical order were required.

If there are better ideas to resolve these, the on-disk definition could be updated in the future.

Extended Attributes (Xattrs)

EROFS supports Extended Attributes (xattr(7)) which are `name:value` pairs associated permanently with inodes since the initial Linux 5.4 version.

Superblock Fields for Xattr Support

The core superblock format is defined in *Superblock*. This section lists the extended fields dedicated to the xattr features.

Offset	Size	Type	Name	Description
0x08	4	u32	feature_co	Compatible feature flags; see *Xattr Feature Flags*
0x2C	4	u32	xattr_blka	Start block address of the shared xattr area; see *Shared Xattr Area*
0x50	4	u32	feature_in	Incompatible feature flags; see *Xattr Feature Flags*
0x5B	1	u8	xattr_pref	Number of long xattr name prefixes; valid when EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES is set; see *Long Xattr Name Prefixes*
0x5C	4	u32	xattr_pref	Location of the long xattr prefix table; valid when EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES is set; see *Long Xattr Name Prefixes*
0x60	8	u64	packed_nid	Packed inode NID. Relevant when long xattr prefixes are embedded in the packed inode's data region
0x68	1	u8	xattr_filt	Must be 0 for the xattr Bloom filter to operate; see *Xattr Filter*
0x69	1	u8	ishare_xat	Long-prefix table index used by image-share xattrs; valid when both EROFS_FEATURE_COMPAT_ISHARE_XATTRS and EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES are set; see *Image-share Xattrs*
0x80	8	u64	metabox_ni	Metabox inode NID. Relevant when shared xattrs or long xattr prefixes are stored in the metabox inode's data region

Xattr Feature Flags

The following superblock feature bits are directly relevant to xattr support.

`feature_compat` Bits

Bit mask	Name	Description
0x0000	EROFS_FEATURE_COMPAT	Enables the per-inode xattr Bloom filter described in *Xattr Filter*
0x0000	EROFS_FEATURE_COMPAT	Stores the shared xattr area in the metabox inode's decoded data region instead of at xattr_blkaddr
0x0000	EROFS_FEATURE_COMPAT	Stores the long xattr prefix table as a standalone region; valid when EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES is set; see *Long Xattr Name Prefixes*
0x0000	EROFS_FEATURE_COMPAT	Enables image-share xattrs. ishare_xattr_prefix_id is valid only when this bit and EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES are both set; see *Image-share Xattrs*

`feature_incompat` Bits

Bit mask	Name	Description
`0x0000004`	`EROFS_FEATURE_INCOMPAT_XATTR_PR`	Enables the long xattr prefix table described in *Long Xattr Name Prefixes*

Inode Fields for Xattr Support

The full compact and extended inode layouts are defined in *Inodes*. For xattr support, both inode variants share the same feature-specific field:

Off-set	Size	Type	Name	Description
0x02	2	`u16`	`i_xattr_icour`	When non-zero, the inline xattr region size is `(i_xattr_icount - 1) * 4 + 12` bytes

Two storage classes exist:

- **Inline xattrs**: stored directly in the metadata block immediately following the inode body (and any inline data tail). They are private to the inode and encoded as a sequence of xattr entry records within the inline xattr region described by `i_xattr_icount`.
- **Shared xattrs**: stored once in the global shared xattr area. An inode references a shared entry through a 4-byte index stored immediately after the fixed 12-byte inline xattr header. Multiple inodes that carry an identical xattr (same name and value) can reference the same shared entry, avoiding redundant per-inode copies.

To further reduce storage overhead for xattrs whose names share a common prefix (for example `trusted.overlay.*` or `security.ima.*`), EROFS supports a long xattr prefix table (`EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES`). Each prefix entry records a `base_index` that refers to one of the standard short namespace prefixes and an additional infix string. An xattr entry whose `e_name_index` selects a long prefix stores only the suffix that follows the full reconstructed prefix, rather than repeating the prefix in every entry.

Inline Xattr Region Layout

The inline xattr region immediately follows the inode body (at a 4-byte aligned offset). Its total size is `(i_xattr_icount - 1) * 4 + 12` bytes, where 12 is the fixed size of the inline xattr body header. The region is structured as:

1. The *Inline Xattr Body Header* (12 bytes, fixed).
2. `h_shared_count` × 4-byte shared xattr index values.
3. Zero or more *Xattr Entry Record* (inline entries).

Inline Xattr Body Header

Offset	Size	Type	Name	Description
0x00	4	u32	h_name_filt	Inverted Bloom filter over xattr names; valid when EROFS_FEATURE_COMPAT_XATTR_FILTER is set; see *Xattr Filter*
0x04	1	u8	h_shared_cc	Number of shared xattr index entries
0x05	7	u8[]	*reserved*	Reserved; must be 0

Xattr Entry Record

Each inline or shared xattr entry has the following layout:

Offset	Size	Type	Name	Description
0x00	1	u8	e_name_len	Length of the name suffix in bytes
0x01	1	u8	e_name_index	Namespace index (maps to a prefix string); see below
0x02	2	u16	e_value_size	Length of the value in bytes

Immediately following the 4-byte xattr entry header: `e_name_len` bytes of name suffix, then `e_value_size` bytes of value. The entire entry (header + name + value) is padded to a 4-byte boundary.

`e_name_index` Namespace Mapping

Rather than storing the full namespace prefix string in every entry, EROFS encodes the xattr namespace prefix as a 1-byte index. Bit 7 of `e_name_index` is the `EROFS_XATTR_LONG_PREFIX` flag. When set, the lower 7 bits index into the long xattr name prefix table (see *Long Xattr Name Prefixes*). When clear, the full byte selects one of the built-in short namespace prefixes:

Value	Prefix
1	user.
2	system.posix_acl_access
3	system.posix_acl_default
4	trusted.
6	security.

Shared Xattr Area

Normally, the shared xattr area begins at block address `xattr_blkaddr`. Each shared entry is an xattr entry record stored contiguously in this area. An inode references a shared entry by its 32-bit index, stored in the inline xattr region immediately after the 12-byte inline xattr header. The index is a byte offset within the shared area divided by 4.

When `EROFS_FEATURE_COMPAT_SHARED_EA_IN_METABOX` is set, the shared xattr pool is stored in the metabox inode's decoded data region rather than at `xattr_blkaddr`.

Long Xattr Name Prefixes

This section applies when `EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES` is set.

When this feature is set, a table of `xattr_prefix_count` prefix entries is present; see *Prefix Table Placement* for where that table is stored. Each entry has the following fixed header, padded together with the variable-length `infix` payload to a 4-byte boundary:

Offset	Size	Type	Name	Description
0x00	2	`u16`	`size`	Byte length of the following content: `base_index` plus the variable-length `infix` payload
0x02	1	`u8`	`base_index`	Built-in short namespace prefix index (see *e_name_index Namespace Mapping*)

The variable-length `infix` bytes begin at offset `0x03`. Their length is `size - 1`, and they are not null-terminated.

The full reconstructed prefix is the concatenation of the short prefix indicated by `base_index` and the `infix` bytes. An xattr entry using a long prefix stores only the name suffix after the reconstructed prefix; `e_name_len` counts only those suffix bytes.

For example, an xattr named `trusted.overlay.opaque` can be represented with `base_index = 4` (`trusted.`) and `infix = "overlay."`, yielding the full prefix `trusted.overlay.`; the stored name suffix is `opaque` with `e_name_len = 6`.

Prefix Table Placement

The xattr prefix table start offset is recorded in `xattr_prefix_start`. The table may be:

- embedded in the metabox or packed inode's data region when `EROFS_FEATURE_COMPAT_PLAIN_XATTR_PFX` is not set; or
- stored in a standalone region when `EROFS_FEATURE_COMPAT_PLAIN_XATTR_PFX` is set.

Xattr Filter

This section applies when `EROFS_FEATURE_COMPAT_XATTR_FILTER` is set.

When this feature is set, `h_name_filter` in the inline xattr body header holds a 32-bit inverted Bloom filter over the inode's xattr names. Each bit position corresponds to one hash bucket:

- A bit value of **1** guarantees that no xattr present on this inode hashes to that bucket, so the queried name is **definitely absent**.
- A bit value of **0** means a matching xattr **may exist** and a full scan is required.

When `xattr_filter_reserved` in the superblock is non-zero, the Bloom filter is disabled unconditionally for all inodes in the image.

Image-share Xattrs

This section applies when both `EROFS_FEATURE_COMPAT_ISHARE_XATTRS` and `EROFS_FEATURE_INCOMPAT_XATTR_PREFIXES` are set.

When these features are set, the superblock field `ishare_xattr_prefix_id` is valid and identifies an entry in the long xattr prefix table. Regular files may carry an xattr whose name equals the prefix identified by `ishare_xattr_prefix_id` (i.e. `e_name_index` selects that entry and `e_name_len` is 0) and whose value is a SHA-256 content fingerprint in the form `sha256:<hex-digest>`.

This convention enables tools to identify files with identical content across different EROFS images by comparing these fingerprints.

Chunk-based Inode Layout

The chunk-based inode layout splits inode data into fixed-size chunks, each mapped to a contiguous range of physical filesystem blocks. This format supports data deduplication and multi-device storage, allowing efficient data sharing among different inodes and images.

Superblock Extension for Chunk-based Inodes and Multiple Devices

The core superblock format is defined in *Superblock*. This section lists the extended fields dedicated to chunk-based inode support and multi-device addressing features.

Offset	Size	Type	Name	Description
0x50	4	`u32`	`feature_i`	`EROFS_FEATURE_INCOMPAT_CHUNKED_FILE` enables chunk-based inodes. `EROFS_FEATURE_INCOMPAT_DEVICE_TABLE` enables the device table described in *Device Table*
0x56	2	`u16`	`extra_dev`	Number of extra devices in addition to the primary one. Valid when `EROFS_FEATURE_INCOMPAT_DEVICE_TABLE` is set
0x58	2	`u16`	`devt_slot`	Starting slot number of the device table on the primary device. The byte offset is `devt_slotoff * 128`. Valid when `EROFS_FEATURE_INCOMPAT_DEVICE_TABLE` is set

Inode Fields for Chunked Inodes

The full compact and extended inode layouts are defined in *Inodes*. For chunk-based inodes, both inode variants share the same feature-specific fields:

Offset	Size	Type	Name	Description
0x00	2	`u16`	`i_format`	Inode format hints; `EROFS_INODE_CHUNK_BASED` (4) is used for chunked inode mode
0x10	4	`u32`	`i_u`	Chunk info summary described below

Inode Data Layout for Chunked Inodes

The following new data layout of an inode is encoded in bits 1–3 of `i_format`.

`EROFS_INODE_CHUNK_BASED` (4)

The entire inode data is split into fixed-size chunks, each occupying consecutive physical blocks. Requires `EROFS_FEATURE_INCOMPAT_CHUNKED_FILE`. `i_u` encodes a chunk info summary and an array of per-chunk address entries follows the inode body.

Chunk-based Structures

When the data layout is `EROFS_INODE_CHUNK_BASED`, the `i_u` field (4 bytes at inode offset 0x10) is interpreted as a chunk info summary:

Chunk Info Summary

Bits	Width	Description
0–4	5	`chunkbits`: chunk size = `2^(blkszbits + chunkbits)`
5	1	`EROFS_CHUNK_FORMAT_INDEXES`: entry format selector (see below)
6	1	48-bit layout specific; ignored for basic chunk-based inodes
7–31	25	Reserved; must be 0

Per-chunk extent entries are stored immediately after the core on-disk inode and the inode xattr region. The number of entries is `i_size / chunk_size`.

Chunk Entry Formats

The `EROFS_CHUNK_FORMAT_INDEXES` bit in the chunk info summary selects one of two per-chunk entry formats:

Block Map Entry (4 bytes)

When `EROFS_CHUNK_FORMAT_INDEXES` is not set, each chunk is described by a single 32-bit block address entry.

Without a device table, this entry is a primary-device block address. With `EROFS_FEATURE_INCOMPAT_DEVICE_TABLE`, it is interpreted in the unified address space and can resolve to either the primary or an extra device.

Offset	Size	Type	Name	Description
0x00	4	`u32`	`startblk`	Starting block address of this chunk

Chunk Index Entry (8 bytes)

When `EROFS_CHUNK_FORMAT_INDEXES` is set, each chunk is described by an 8-byte record that supports multi-device addressing.

Offset	Size	Type	Name	Description
0x00	2	`u16`	*dontcare*	48-bit layout specific; ignored for basic chunk-based inodes
0x02	2	`u16`	`device_i`	Device selector. `0` uses unified-address-space resolution; non-zero values directly select extra devices
0x04	4	`u32`	`startblk`	32-bit starting block address; interpretation depends on `device_id`

When `device_id` is `0`, `startblk` is interpreted exactly like a *Block Map Entry (4 bytes)*: it is an absolute block address in the unified address space. When `device_id` is non-zero, it directly selects an extra device and `startblk` is interpreted as a block address on that device. See *Block Address Resolution for Chunk-based Inodes*.

Multi-device Support

Device Table

This section applies when `EROFS_FEATURE_INCOMPAT_DEVICE_TABLE` is set.

When `EROFS_FEATURE_INCOMPAT_DEVICE_TABLE` is set, the `extra_devices` superblock field gives the number of additional block devices. They are described by an array of 128-byte device slot records stored on the primary device. The first record begins at byte offset `devt_slotoff * 128`.

Each record contains:

Offset	Size	Type	Name	Description
0x00	64	`u8[]`	`tag`	User-specific identifier: The kernel never parses it in any form
0x40	4	`u32`	`blocks`	32-bit total block count of this device
0x44	4	`u32`	`uniaddr`	32-bit unified starting block address of this device
0x48	4	`u8[]`	*dontcare*	48-bit layout specific; ignored for basic chunk-based inodes
0x4C	52	`u8[]`	*reserved*	Reserved; must be 0

Block Address Resolution for Chunk-based Inodes

Chunk Index Entry

When the chunk index entry format is used, `device_id` controls how `startblk` is interpreted:

- `device_id = 0`: `startblk` is an absolute block address in the unified address space; see *Block Map Entry* for details.
- `device_id = N` (1 N `extra_devices`): `startblk` is a block address on extra device N, whose record begins at byte offset `(devt_slotoff + N - 1) * 128` on the primary device.

Block Map Entry

The simple block map entry format has no `device_id` field. Instead, `startblk` is an absolute block address in the unified address space, and the reader uses `uniaddr` from the device table to identify the target device: it finds the slot `i` whose range `[uniaddr[i], uniaddr[i] + blocks[i])` contains `startblk`, then derives the intra-device block address as `startblk  uniaddr[i]`. Chunk index entries with `device_id = 0` use this same interpretation.

5.4.2 Native sub-filesystem merging

In addition to the OSTree/Composefs file-based model, EROFS itself supports three native forms to merge common external binary sources (like sub-filesystem layers) into one filesystem and mount the merged filesystem in one go.

Single device

Unlike other kernel filesystems which have inflexible layouts, EROFS has only one fixed-offset on_disk part: *the superblock*. It's quite easy to compose external payloads such as binary formats like tarballs or other filesystems (e.g., EROFS, EXT4, XFS, etc.) just with **linear physical mappings** to form the entire filesystem view.

It's particularly useful for virtualization cases since *only one EROFS mount* will be passed in via virtio-blk or virtio-pmem instead of individual parts.

Especially for each virtio-pmem instance, the host page cache can be easily shared at a finer sub-image granularity due to linear physical mappings compared to other approaches, which are typically based on traditional disk snapshots.

Tip: This way has already been usable since the initial EROFS version. However, if your use cases also cover multiple devices, an on-disk device table will be needed then; See below.

Multiple devices / blobs

Since Linux 5.16, EROFS has supported multiple devices explicitly as its external data sources. EROFS device table can indicate external data on extra devices as below:

Besides, it is quite useful for scenarios where we have no way to compose a single specific block device from individual sources as described in the previous section.

In addition, since Linux 5.19, EROFS has added **EROFS over fscache** to make use of fscache caching framework without explicit block devices, which is currently used by Dragonfly Nydus image service. There are some benefits such as massive blobs and on-demand loading. For more details, please look into this.

Single flattened device

EROFS can still be mounted as a single, flattened device even if the on-disk device table exists. It's **highly recommended** that an on-disk device table is always included in the long term for related use cases.

CHAPTER

SIX

DEVELOPER GUIDES

6.1 Git Repositories

6.1.1 erofs-utils

erofs-utils is developed with Git, and multiple branches are available for different needs:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/xiang/erofs-utils.git
```

branch	description	rebase?
dev	erofs-utils development tree	Maybe
experimental	erofs-utils tree with unstable patches for testing	Yes
master	erofs-utils stable tree	No

6.1.2 Linux kernel source

If you're interested in EROFS kernel development, it is recommended to keep your local code in sync with the latest EROFS development repo:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/xiang/erofs.git
```

branch	description	rebase?
fixes	EROFS kernel fixes-only tree (for this cycle)	Yes
dev	EROFS kernel development tree (for the next cycle)	Yes

6.2 Mailing List

EROFS has its own development mailing list hosted by OzLabs: <linux-erofs@lists.ozlabs.org>

You can subscribe to the mailing list to receive the latest status of EROFS.

When posting, it is helpful to:

- Add an additional tag in the subject like `[PATCH]`, `[question]` or `[bug report]`, etc.;
- Avoid top-posting if possible.

All patches should follow the Linux kernel's coding style. Additionally, as one of the Linux kernel development communities, patches require the "sign-off" procedure.

The sign-off should be appended as a simple line at the end of the commit message for the patch, which claims that you agree to Developer Certificate of Origin. In other words, it certifies that either you wrote it or have the right to pass it on as an open-source patch.

Then you just add a line saying:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

using your real name (sorry, no pseudonyms or anonymous contributions.)

6.3 Matrix room

You can also join our Matrix room (#erofs:matrix.org) for informal discussions.

However, proposals, bug reports, and/or patches should be sent to the mailing list eventually so that they can be properly archived.

CHAPTER

SEVEN

ROADMAP

Note that items marked with **[GSoC]** are also intended for Google Summer of Code (GSoC) projects.

7.1 Linux Kernel

- EROFS page cache sharing across different filesystems;
- Preliminary EROFS Rust in-kernel adaption (EXPERIMENTAL, program for students);

7.2 Userspace tools (erofs-utils)

- Stabilize liberofs APIs;
- Virtual machine template feature support (memory + disk snapshots can be incrementally packed into multi-blob EROFS images);
- **[GSoC]** *Multi-threaded decompression*;
- Rebuild improvements (including incremental updates [1]).

7.3 Containers

- (containerd) EROFS Support and Improvements;
- (Kata Containers) EROFS Snapshotter Support in Kata;
- (gVisor) EROFS Snapshotter Support.

7.4 Miscellaneous items

- **[GSoC]** *Porting EROFS to BSD Kernels*.

CHAPTER

EIGHT

GOOGLE SUMMER OF CODE 2026

8.1 About Google Summer of Code

The Google Summer of Code is a global, online program focused on bringing new contributors into open source software development. Contributors work with an open source organization on a 12+ week-long program under the guidance of mentors.

8.2 Project Proposal Guidelines

Contributors/applicants are responsible for writing a proposal and submitting it to Google before the application deadline.

See the Project Ideas for a starting point for project ideas. We welcome proposals which are variations of these project ideas and new project ideas as well. Please reach out to us on the development mailing list or Matrix room to discuss project proposals.

In order to ensure the projects run healthily, your proposal must contain a clear yes/no statement about whether you used LLM-based AI tools (ChatGPT, Gemini, etc.) to help you write it. Include exactly one of the following statements in your proposal:

```
This proposal was written with the assistance of [ChatGPT/Gemini/etc] to [check spelling
/ check accuracy / format text].
```

or:

```
This proposal was written without the use of any AI tools.
```

If your proposal does not contain either of these statements, your proposal will not be considered.

Using AI tools for refining project proposals is allowed, but please try to keep the proposal within your own development capability.

Please **do not submit a proposal completely generated by AI** since it's just a waste of our time.

8.3 Project Ideas

8.3.1 Multi-threaded Decompression Support in fsck.erofs

Proposed mentors: Yifan Zhao (@SToPire), Chunhai Guo (@speedan1), Gao Xiang (@hsiangkao)
Languages: C
Estimated project length: 350 hours
Difficulty: hard
Skills:

- Proficiency in C programming;
- Experience with multi-threaded programming;
- Experience with file system concepts and operations.

Description

EROFS is designed for modern high-performance immutable use cases and is widely used in OS images such as Android system images and container images. Its userspace tool, fsck.erofs, is critical for filesystem integrity checking, image unpacking, and regression testing.

Currently fsck.erofs is strictly single-threaded, therefore it unpacks each file one-by-one, even though:

- EROFS compression units (e.g., pclusters) are largely independent;
- Modern computer systems provide abundant CPU parallelism;
- Unpacking workloads for compressed files are often CPU-bound rather than I/O-bound.

For large images, extraction time scales poorly, which negatively impacts image installation, CI pipelines and/or developer workflows. Therefore, introducing multi-threaded decompression would unlock significant performance improvements of these use cases.

The primary goal is to design and implement a new multi-threaded decompression pipeline for fsck.erofs that is efficient, safe and maintainable.

Specific goals:

- Parallelized decompression at different levels (compressed clusters, inodes, etc.)
- Always perform better than the single-threaded approach among all supported algorithms (LZ4, LZMA, DEFLATE and Zstandard);
- Maintain correctness and stability;
- Minimize memory overhead and contention;
- The unpacking performance of metadata compressed filesystems should be greatly improved too;
- Add tests to look after this new feature (e.g. stress tests).

8.3.2 Support generating filesystems from manifests with mkfs.erofs

Proposed mentors: Chengyu Zhu (@ChengyuZhu6), Gao Xiang
Languages: C
Estimated project length: 175 hours
Difficulty: medium
Skills:

- Proficiency in C programming;
- Experience with file system concepts and operations.

Description

For local builds, currently mkfs.erofs only supports generating from source directories or tarballs. However, building from source directories has the following disadvantages:

- Limited inode metadata control;
- Requires privileges for special inodes;
- No explicit data dump ordering control;
- Slower for very large filesystem trees.

The primary goal is to implement the functionality to generate EROFS images from manifest files. Note that manifest formats are highly fragmented, so you should implement support for at least two common manifest formats, for example:

- composefs-dump(5);
- Modified unix proto files;
- BSD mtree(5).

Dedicated test cases should be added to ensure its correctness.

8.3.3 Complete Filesystem Feature Support for erofs-rs

Proposed mentors: @Dreamacro, Gao Xiang
Languages: Rust
Estimated project length: 350 hours
Difficulty: hard
Skills:

- Proficiency in Rust programming;
- Experience with file system concepts and operations.

Description

EROFS is a high-performance, space-efficient read-only filesystem widely used in the Linux ecosystem.

Currently, there is no mature, full-featured EROFS implementation in pure Rust. However, erofs-rs has the following features that make it particularly appealing:

- `no_std` support for embedded systems;
- Zero-copy parsing via mmap (std) or byte slices (no_std).

However, it currently only includes a reader, and even this reader lacks the following basic features:

- Support for extended attributes (xattrs);
- Support for multiple devices;

- Support for two compression layouts;
- Support for 48-bit addressing layouts;
- Support for metadata compression.

The primary goal is to complete the erofs-rs feature set, including the missing reader features listed above, and additionally implement a writer with an efficient block allocator as a plus.

8.3.4 Porting EROFS to BSD Kernels (FreeBSD Focus)

Proposed mentors: Gao Xiang, Hongbo Li (@hb-lee)
Languages: C
Estimated project length: 350 hours
Difficulty: hard
Skills:

- Proficiency in C programming;
- Experience with file system concepts and operations.

Description

EROFS is a high-performance, space-efficient read-only filesystem widely used in the Linux ecosystem, particularly for immutable infrastructure, containers, and embedded systems.

Today, EROFS is heavily used in:

- Android system images;
- Container images and snapshotters (containerd, gVisor, Kata containers);
- Immutable OS (AWS Bottlerocket) and cloud infrastructure.

However, BSD kernels currently lack native implementation for EROFS, forcing interested users to rely on other alternatives. Porting EROFS to BSDs would:

- Enable direct mounting of native EROFS images;
- Improve interoperability between Linux and BSD systems;
- Provide FreeBSD with a modern high-performance immutable filesystem optimized for container and immutable OS workloads.

The primary goal is to implement EROFS filesystem support in the FreeBSD kernel, following FreeBSD VFS conventions while remaining compatible with the standard EROFS on-disk format.

Key objectives:

- Implement a FreeBSD kernel EROFS filesystem driver;
- Support mounting and reading standard EROFS images;
- Integrate EROFS with FreeBSD's VFS, buffer cache, and VM systems;
- Validate correctness and performance using real-world workloads;
- Lay groundwork for future BSD ports (OpenBSD, NetBSD).

8.3.5 Advanced Fuzzing and Image Injection for the Kernel and erofs-utils

Proposed mentors: Yifan Zhao, Hongbo Li, Gao Xiang
Languages: C, Go and/or Rust
Estimated project length: 175 hours
Difficulty: medium
Skills:

- Proficiency in C programming;
- Experience with file system concepts and operations;
- Familiarity with fuzzing frameworks (e.g., AFL++, libFuzzer) is a plus.

Description

EROFS aims to be a secure, immutable image-based kernel filesystem by design. Because its on-disk format contains less redundant metadata and is designed to tolerate bogus or corrupted values, EROFS behaves differently from generic writable filesystems. In addition, its immutable design means that all writable data is copied up (aka copy-on-write) into another local trusted filesystem. This makes it safer than writing directly to an untrusted and potentially inconsistent generic writable filesystem.

We pay particular attention to the EROFS core on-disk format. Although the format design is simple and the implementation (especially for the core format) is straightforward, it is highly beneficial to develop more advanced tools alongside the current syzkaller and the existing erofs-utils fuzzer. These tools will keep the codebase robust and allow us to address random human-introduced bugs more actively and in time.

The main goal is to implement an advanced fuzzing tool and an image injection tool. These tools may be easier to implement using go-erofs (Go) or erofs-rs (Rust), for example. We also intend to enable a new GitHub Actions CI workflow to perform periodic fuzzing.

This will allow us to maintain the kernel and erofs-utils implementations in better shape.

CHAPTER

NINE

FREQUENTLY ASKED QUESTIONS

9.1 Why are images packaged in EROFS larger than those with SquashFS?

First of all, the initial target use cases of EROFS are *high-performance embedded scenarios, such as smartphones powered by Android*. Runtime performance is always the top priority for EROFS (or, systems and applications will be lagged), even if it means sacrificing some ultra-space savings to avoid significant performance regressions against uncompressed approaches.

However, EROFS has landed **several new on-disk features** to narrow the slight size difference with SquashFS or even outperform SquashFS. When comparing, please ensure the same configuration is used:

- **Compression algorithm (if data is compressed)**: EROFS uses *LZ4* by default due to lowest decompression latencies among popular open-source algorithms, while SquashFS uses *GZIP* instead;
- **Compressed extent size**: Almost all filesystems that natively support compression typically cut data into compressed extents for random access. EROFS focuses on smaller physical clusters to maximize random performance and use *block-sized physical clusters* by default (usually 4 KiB), whereas SquashFS uses *128 KiB*. It can be adjusted using the `-C` option with `mkfs.erofs`.

Note: Large physical clusters (e.g., 1MiB) can significantly degrade random read performance as well as increase memory usage. Please conduct a thorough evaluation by factoring in the target image size prior to deployment.

- **Compression level**: For example, EROFS uses *LZ4HC_CLEVEL_DEFAULT (level 9)* for LZ4 HC, whereas SquashFS often uses *LZ4HC_CLEVEL_MAX (level 12)*;
- **Advanced features**: Enable the `-Efragments` option for EROFS when comparing with SquashFS.

In addition, EROFS may produce larger images due to the following differences:

- **Inode size**: Currently EROFS has to use 64-byte on-disk inodes (extended inodes) to support per-inode nanosecond timestamps, whereas SquashFS often uses 32-byte inodes with only 4-byte timestamps for regular files; Consider switching to 32-byte EROFS compact inodes (e.g., by using `-T`) if per-file timestamps are not a strong requirement;
- **Metadata compression**: If your test sets contain a large number of files, EROFS may result in larger images compared to SquashFS if metadata compression is not enabled. Optional metadata compression has been supported since Linux 6.17; consider enabling it if metadata size is a concern.
- **File-based deduplication**: SquashFS deduplicates files with identical data by default (it can be disabled with `-no-duplicates`), whereas EROFS does not (except for hardlinks). However, EROFS offers the similar functionality using `-Efragments` and even finer-grained data deduplication using `-Ededupe`.

- **BCJ filters**: SquashFS can compress with XZ algorithm of BCJ enabled to optimize executable code. This feature is not supported by EROFS for now, but there are plans to introduce BCJ filters for all EROFS-supported algorithms.

Note that EROFS is still under active development. The features mentioned above are not top priorities at the moment due to limited development resources (anyway, SquashFS has existed for over 20 years) and target use scenarios, but they will be considered in the future, and contributions are always welcome. Again, note that SquashFS doesn't always outperform EROFS in image size either. EROFS images are often significantly smaller (while still offering better runtime performance) when compressing files in small compressed extent sizes (especially smaller than 32KiB).

Additionally, EROFS has supported CDC-like *compressed data deduplication* since Linux 6.1, which also gives extra space savings (although `-Ededupe` is still single-threaded).

In brief, if image size is your top priority, please ensure that the options `-Ededupe` and `-E(all-)fragments` are specified with mkfs.erofs. At least, enable `-Efragments` to match the default configuration of SquashFS.

9.2 Under construction..

CHAPTER

TEN

CONTRIBUTORS

EROFS has been contributed by community developers around the world, including but not limited to (in alphabetical order):

- By vendor (raw)
  - Alibaba Group (@linux.alibaba.com)
  - ByteDance (@bytedance.com)
  - Coolpad (@coolpad.com, @yulong.com)
  - Docker (@docker.com)
  - Google (@google.com)
  - Huawei (@huawei.com)
  - Inspur (@inspur.com)
  - NVIDIA (@nvidia.com)
  - OPPO (@oppo.com)
  - Red Hat (@redhat.com)
  - Shanghai Jiao Tong University (@sjtu.edu.cn)
  - Sony Group Corporation (@sony.com)
  - South China University of Technology (@mail.scut.edu.cn)
  - Tencent (@tencent.com)
  - Tuxera (@tuxera.com)
  - Uniontech (@uniontech.com)
  - Vivo (@vivo.com)
  - Xiaomi (@xiaomi.com)
- Individuals

Since it's *an incomplete list*, feel free to submit a pull request to add yourself here if you'd like to be highlighted as an EROFS contributor.