Reproducible XFS Filesystem Images

No Mount Required

Creating bit-for-bit identical filesystem images at mkfs time

Luca Di Maio

FOSDEM 2026

talk@fosdem:~$ whoami

Luca Di Maio

- SWE @Chainguard
- Open Source Enthusiast
- **89luca89**
- ✉ luca.dimaio1@gmail.com

Why Reproducible Filesystems?

Supply Chain Security

Verify builds have not been tampered with: if source is identical, output must be identical

Auditability

Anyone can rebuild and verify: no trust required in build infrastructure

Debugging

Reproduce exact conditions: bisect issues across builds with confidence

Distribution Trust

Users can verify distro images match published sources

The Problem

Traditional workflow:

```
mkfs.xfs /dev/sdX → mount → cp -a rootfs/* /mnt → umount
```

- Requires root privileges for mounting
- Timestamps vary with build time
- Random seeds lead to different inode generation numbers

Failed Attempt: libfaketime

Good news: Empty filesystems CAN BE reproducible!

```
$ LD_PRELOAD=libfaketime.so FAKETIME="1970-01-01 01:00:00" \
    mkfs.xfs -m uuid=44444444-4444-4444-4444-444444444444 test1.img
$ LD_PRELOAD=libfaketime.so FAKETIME="1970-01-01 01:00:00" \
    mkfs.xfs -m uuid=44444444-4444-4444-4444-444444444444 test2.img
$ md5sum *.img
ed581f44068a6ab94062a65d5272e1f2  test1.img
ed581f44068a6ab94062a65d5272e1f2  test2.img  # ← Identical!
```

Bad news: Populating via mount breaks reproducibility!

```
$ mount -o noatime,nodiratime test1.img /mnt
$ tar --no-atime -xf rootfs.tar -C /mnt && umount /mnt
# ... repeat for test2.img ...
$ md5sum *.img
a1b2c3d4...  test1.img
e5f6g7h8...  test2.img  # ← Still different every time!
```

Failed Attempt: Why It Fails

The kernel ignores userspace tricks

LD_PRELOAD is userspace-only

- Kernel syscalls bypass libfaketime
- inode ctime set by kernel, not libc

AG allocation non-deterministic

- which AG gets the inode
- Parallel/async I/O leads to non-deterministic allocation order
- B+tree shape depends on order

What Others Do

reproducible-builds.org/docs/system-images/ documents filesystem support:

Filesystem	Support
ext2/ext3/ext4	`mkfs.ext{2,3,4}` with `-E hash_seed`, `-U` and `-d` to populate
SquashFS	`mksquashfs` has `-repro` flag
EroFS	`mkfs.erofs` with `-T` for timestamps and `-U` for uuid
ISO	`xorriso` supports `SOURCE_DATE_EPOCH` + `--modification-date`
XFS	No support... **until now!**

> **Common pattern:** populate filesystem at creation time, control all sources of non-determinism, avoid mounting.

The Path Forward

Port the same approach to XFS:

1. Directory Population (`-p`)

- Populate filesystem from directory tree at mkfs time
- All file types, xattrs, hardlinks, timestamps

2. Deterministic Seed/Time

- `SOURCE_DATE_EPOCH`: fixed timestamps
- `DETERMINISTIC_SEED=1`: fixed seed (`0x53454544`)

3. Tests

- Verifies bit-for-bit reproducibility
- Uses fsstress to generate realistic content

Patch 1 - Populating

Patch 1: Why Not Just Extend Protofile?

XFS already has protofile support (`-p file`). Extend it?

```
/
0 0
d--755 1000 1000
: Descending path test/
 a d--755 1000 1000
  file1 ---644 1000 1000 test/a/file1
 $
 b d--755 1000 1000
  file2 ---644 1000 1000 test/b/file2
 $
 c d--755 1000 1000
  d d--755 1000 1000
  $
 $
$
```

Problems:

- **Limited xattr support**

> regular files only

- **No timestamps**

> always using current time, not source time

- **Backward compatibility**

> changing format might break existing users

Patch 1: Architecture Overview

> **Idea**: mkfs.xfs in userspace uses libxfs which allocates inodes sequentially. Without kernel-level parallelism, allocation order becomes deterministic.

1. **`setup_proto()`**
   - Regular file → existing protofile logic
   - Directory → new `populate_from_dir()` logic
2. **`populate_from_dir()`**
   - Recursively traverse source dir and populate new FS
3. **`handle_direntry()`**
   - Dispatches on file type (`st_mode`):
     - `S_IFDIR` → `create_directory_inode()` → recurse
     - All others → `create_nondir_inode()` (files, symlinks, devices, fifos, sockets)

Patch 1: Timestamps

Files have multiple timestamps (atime, mtime, ctime, crtime).

Which ones should we preserve from the source, and which should we set to "now" (potentially `SOURCE_DATE_EPOCH`)?

Timestamp	Behavior	Rationale
mtime	Always copied	Meaningful modification time
atime	Optional (`-p dir,atime=1`)	Usually noise
ctime/crtime	`current_time()`	New inode creation

> **Note:** Most reproducible builds strip atime anyway; option exists for edge cases.

Patch 1: Attributes

File descriptor challenge: Different types need different `open()` flags:

- Regular files: `O_RDONLY | O_NOATIME` (need to read content)
- Symlinks/sockets/FIFOs: `O_PATH` only (can't open for reading)
  - `fgetxattr()`/`flistxattr()` fail with `EBADF` on `O_PATH` fds.
  - Fall back to `lgetxattr()`/`llistxattr()` (path-based) when `errno == EBADF`.

> Symlinks have xattrs too (SELinux labels). Without this fallback, labels get lost.

Patch 1: Hardlinks & Resource Management

Hardlink tracking: Maps `src_ino → dst_ino` in dynamic array.

- First encounter (`st_nlink > 1`): Create file normally, store mapping
- Subsequent: Look up `dst_ino`, create dir entry, call `libxfs_bumplink()`
- Growth: 2x for small arrays, +25% for large (threshold: 1024)

Performance:

- Linear O(n) — tested 1.3M inodes, 400k hardlinks in seconds. I/O bounded; good enough.

Patch 2 - Determinism

Patch 2: Populating is not enough

Patch 1 populates from directory, but `ctime`/`crtime` use `gettimeofday()` and inode generation comes from `getrandom()` → **not reproducible**

What about libfaketime?

```
~$ LD_PRELOAD=libfaketime.so \
   FAKETIME="1970-01-01 01:00:00" FAKERANDOM_SEED="0x12345678aaaabbbb" \
   mkfs.xfs \
     -m uuid=44444444-4444-4444-4444-444444444444 \
     -p ./rootfs disk.img
```

It works but there are problems

- Fragile: `LD_PRELOAD` breaks with setuid binaries, static linking
- Non-standard: doesn't follow `SOURCE_DATE_EPOCH` convention
- Requires additional tooling for `getrandom()` interception

Patch 2: Reproducibility Environment Variables: SOURCE_DATE_EPOCH

`SOURCE_DATE_EPOCH` - Fixed timestamps

- Standard from `reproducible-builds.org`
- Unix timestamp (seconds since 1970)
- Used for `ctime` and `crtime` of all new inodes
- Example: `SOURCE_DATE_EPOCH=1234567890`

```
bool current_fixed_time(struct timespec64 *tv) {
[ ... ]
    if (!read_env) {
        read_env = true;
        char *sde = getenv("SOURCE_DATE_EPOCH");
        if (sde && sde[0] != '\0') {
            errno = 0;
            epoch = strtoll(sde, &endp, 10);
[ ... ]
        }
    }
[ ... ]
    if (read_env && enabled) {
        tv->tv_sec = epoch;
        tv->tv_nsec = 0;
        return true;
    }
[ ... ]
}
```

Patch 2: Reproducibility Environment Variables: DETERMINISTIC_SEED

`DETERMINISTIC_SEED=1` - Fixed random seed

- Replaces `getrandom()` with fixed value `0x53454544` ("SEED" in ASCII)
- Affects inode generation numbers

> **Is it safe?** Inode generation numbers (`di_gen`) change when an inode number is reused after deletion. At mkfs time this is not relevant.

```
bool get_deterministic_seed(uint32_t    *result) {
[ ... ]
    if (!read_env) {
        read_env = true;
        seed_env = getenv("DETERMINISTIC_SEED");
        if (seed_env && strcmp(seed_env, "1") == 0)
            enabled = true;
    }
    if (read_env && enabled) {
        *result = deterministic_seed;
        return true;
    }
    return false;
}
```

Patch 3 - testing

Patch 3: testing

Function	Purpose
`_create_proto_dir()`	`fsstress -n 2000 -p 2`, mkfifo, mknod (blk/chr), af_unix socket
`_check_mkfs_xfs_options()`	Check `-m uuid=` support, `-p` populate support, grep `SOURCE_DATE_EPOCH` in binary

```
_mkfs_xfs_reproducible() {
    SOURCE_DATE_EPOCH=$FIXED_EPOCH \
    DETERMINISTIC_SEED=1 \
    $MKFS_XFS_PROG -f -m uuid=$FIXED_UUID -p "$PROTO_DIR" "$img"
}

# Run 3 iterations, compare SHA256 hashes
hash1=$(_run_iteration 1)
hash2=$(_run_iteration 2)
hash3=$(_run_iteration 3)

[ "$hash1" = "$hash2" ] && [ "$hash2" = "$hash3" ]  # Must match!
```

In Action

In Action: usage examples

Basic: populate from directory

```
mkfs.xfs -p ./rootfs disk.img
```

Preserve access timestamps from source

```
mkfs.xfs -p ./rootfs,atime=1 disk.img
```

In Action: usage examples

Reproducible filesystem

> avoid the use of libfaketime

```
~$ truncate -s 1G test{1,2}.img
~$ md5sum *.img
cd573cfaace07e7949bc0c46028904ff  test1.img
cd573cfaace07e7949bc0c46028904ff  test2.img
~$ SOURCE_DATE_EPOCH=1234567890 DETERMINISTIC_SEED=1 \
  mkfs.xfs -m uuid=12345678-1234-1234-1234-123456789abc test1.img
[ ... ]
~$ SOURCE_DATE_EPOCH=1234567890 DETERMINISTIC_SEED=1 \
  mkfs.xfs -m uuid=12345678-1234-1234-1234-123456789abc test2.img
[ ... ]
~$ md5sum *.img
1facbedb3391cddce08c30c576e8860c  test1.img
1facbedb3391cddce08c30c576e8860c  test2.img
```

> **Still required:** `-m uuid=...`
> UUID is still generated randomly by default

In Action - mkosi

```
#!/bin/sh

mkosi \
  -d debian -r bookworm \
  -p base-files \
  -p dbus \
  -p systemd \
  -t directory \
  -o rootfs\
  --seed aaaabbbb-aaaa-bbbb-aaaa-bbbbaaaabbbb \
  --source-date-epoch 1234567890 \
  --remove-files /var/cache/ldconfig/aux-cache \
  --remove-files /var/log/alternatives.log

truncate -s 1G rootfs.img

DETERMINISTIC_SEED=1 SOURCE_DATE_EPOCH=1234567890 \
  mkfs.xfs \
    -m uuid=12345678-1234-1234-1234-123456789abc \
    -p ./rootfs/ rootfs.img
```

In Action - mkosi

Run this script a couple of times

```
~$ ./build.sh
~$ ls
build.sh  rootfs  rootfs.img
~$ mv rootfs.img rootfs-og.img && rm -rf rootfs
~$ ls
build.sh  rootfs-og.img
~$ ./build.sh
~$ ls
build.sh  rootfs  rootfs.img rootfs-og.img
~$ file *.img && md5sum *.img
rootfs-og.img: SGI XFS filesystem data (blksz 4096, inosz 512, v2 dirs)
rootfs.img:    SGI XFS filesystem data (blksz 4096, inosz 512, v2 dirs)
715a2e67b66e5571b16fca715a70bc47  rootfs-og.img
715a2e67b66e5571b16fca715a70bc47  rootfs.img
```

Use Cases

- **Distributions** -> Verifiable cloud/VM images, reproducible installation media
- **Embedded Systems** -> Unprivileged rootfs generation, Yocto/Buildroot/CI friendly
- **Security** -> Supply chain verification, SLSA compliance, build attestation
- **Testing** -> Deterministic test fixtures, reproducible bug reports

Questions?

Luca Di Maio

luca.dimaio1@gmail.com

- `mkfs` patches:
  - https://lore.kernel.org/linux-xfs/20250730161222.1583872-1-luca.dimaio1@gmail.com/
  - https://lore.kernel.org/linux-xfs/20251108143953.4189618-1-luca.dimaio1@gmail.com/
- Test patches: https://lore.kernel.org/linux-xfs/20260108142222.37304-1-luca.dimaio1@gmail.com/
- Links
  - https://reproducible-builds.org/docs/system-images/