Revamping rfs

Recent enhancements to radare2's filesystem library: UBIFS support, interactive navigation, and advanced metadata extraction for analysis and forensics

Miquel Sancho **MiKi** (mikelloc)

October 2025

About

Miquel S. a.k.a **MiKi** (mikelloc)

Background:

- Telecommunications Engineer (CS+EE)
- Firmware, device drivers, industrial control, RF.
- Focus on firmware analysis and embedded systems/computer architectures.
- Now working as software engineer on backend and payments processing.

Contact:

- @mikelloc (X)
- @mikelloc@infosec.exchange (Mastodon)

Overview: What We'll Cover Today

Enhancements:

1. **UBIFS Support** - Embedded Linux filesystem analysis
2. **Filesystem Metadata (`mn`)** - Extract volume info, timestamps, geometry
3. **rafs2 Tool** - Standalone filesystem manipulation
4. **Miknight Commander** - Interactive TUI file manager
5. ~~**SSHFS Plugin** - Remote filesystem access via SSH~~ (sssh! surprise!)
6. **Notes on architecture and contributing**

Rationale / History

But prior to start some history

- radare started as a forensics recovery tool.
- Had much more evolution around disassembling, decompiling.

Understanding UBI

UBI = Unsorted Block Images

Purpose: Essential component for NAND flash management

- Wear-leveling: Distributes writes evenly across the chip
- Bad block management: Handles defective blocks gracefully
- Bit flip handling: Corrects errors that occur in NAND

Position in Stack: Sits between MTD (Memory Technology Device) and filesystem layer

UBI Stack Architecture

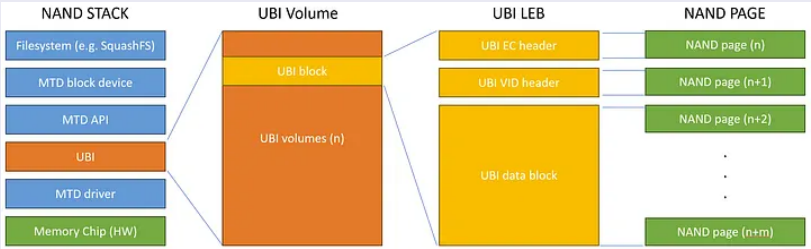


Figure 1: NAND stack using UBI

UBI Volumes:

- Up to 128 user volumes (actual data storage)
- Up to 4096 internal volumes (housekeeping, volume tables)

LEB vs PEB:

- PEB (Physical Erase Block): NAND flash block (e.g., 128 KiB)
- LEB (Logical Erase Block): UBI block, slightly smaller (e.g., 126 KiB)
- LEB has headers: EC (Erase Counter) + VID (Volume Identifier) + data

UBI Block Allocation

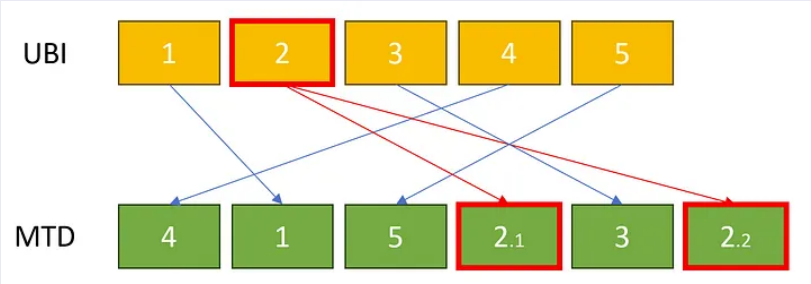


Figure 2: UBI LEB allocation of MTD PEBs

Key Concept: LEBs are scattered, not sequential!
Challenges:

- LEBs spread across NAND for wear-leveling
- VID header tracks block sequence numbers
- Duplicate blocks possible (with different sequence numbers)
- Each LEB consists of multiple NAND pages

Extraction Process:

1. Parse all UBI LEBs from MTD dump
2. Rearrange by logical order (VID headers)
3. Check sequence numbers for duplicates
4. Reconstruct original volume

UBIFS Support

Let's make radare speak UBIFS!

Why UBIFS Matters

UBIFS = Unsorted Block Images File System

- Standard for embedded Linux devices (routers, IoT, medical devices)
- No native FUSE support on many systems
- Existing tools only extract, don't analyze in-place

Use Cases:

- Firmware security analysis
- IoT device reverse engineering
- Embedded system forensics

UBIFS Architecture

```
UBI Layer (Unsorted Block Images):
|-- Erase Counter Headers: Track block wear for even distribution
|-- Volume ID Headers: Identify and manage logical volumes
+-- Volume Table: Stores metadata for all volumes

UBIFS Layer (filesystem):
|-- Superblock Node: Contains filesystem parameters and geometry
|-- Master Node: Points to root index and maintains statistics
+-- Index B-tree: Organizes filesystem structure
    |-- Inode Nodes: Store file metadata and attributes
    |-- Data Nodes: Hold compressed file content
    +-- Directory Entry Nodes: Map filenames to inodes
```

Key Features:

- Hash table-based inode cache for O(1) lookups
- LZO/ZLIB decompression with raw deflate support
- Full mount, directory listing, and file reading

New Commands: prgr and mis

They came for free while implementing UBIFS
prgr - Print Raw Gzip (deflate):

```
mis /mnt/etc/banner    # Seek to compressed file
b 396                  # Set block size to compressed size (optional)
prgr                   # Decompress raw deflate (no ZLIB headers)
```

Why needed? UBIFS/SquashFS use raw deflate (RFC 1951), not ZLIB (RFC 1950)
mis - Mount Info + Seek:

```
# mi /mnt/etc/passwd; s 0x4ec398  (manual, error-prone)
# mis /mnt/etc/passwd              (automatic, correct)
```

Automatically finds mapped IO address and seeks correctly!

UBIFS Demo

Live demonstration:

```
# Mount UBIFS image
r2 test/bins/fs/ubifs.img
m /mnt ubifs 0

# Navigate filesystem
md /mnt
md /mnt/etc

# Read and decompress file
mi /mnt/etc/banner
mis /mnt/etc/banner
```

Filesystem Metadata

The `mn` Command

Problem: Need metadata from filesystem images

- Volume labels, UUIDs, serial numbers
- Mount times, creation dates, last write
- Block/sector sizes, space usage
- Filesystem-specific features

Solution: `mn` (Mount iNformation) command

Supported: ext2/3/4, FAT12/16/32, NTFS, ISO9660, HFS, UBIFS

Filesystem: The Madness

Each filesystem has its own time representation:

Lasciate ogne speranza, voi ch'entrate *(Abandon all hope, ye who enter here.)*

Filesystem	Epoch	Unit	Range	Notes
Unix/ext2-4	1970-01-01	seconds	1970-2038/2106	time_t (32/64-bit)
FAT	1980-01-01	2 seconds	1980-2107	DOS time, 2-sec precision!
NTFS	1601-01-01	100 nanoseconds	1601-30828	FILETIME, insane precision
HFS	1904-01-01	seconds	1904-2040	Mac epoch
HFS+	1904-01-01	seconds	1904-2040	Mac epoch (UTC)
ISO9660	Year 1	10 milliseconds	0001-9999	Centisecond precision
UBIFS	1970-01-01	seconds + ns	1970-2106	Unix time + nanoseconds

Timestamp Conversion Examples

NTFS FILETIME to Unix:

```
// NTFS: 100-nanosecond intervals since 1601-01-01
// Difference: 369 years = 11,644,473,600 seconds
static time_t ntfs_filetime_to_unix(ut64 filetime) {
    const ut64 ntfs_epoch_offset = 11644473600ULL;
    ut64 seconds = filetime / 10000000ULL;  // 100ns -> seconds
    return (time_t)(seconds - ntfs_epoch_offset);
}
```

FAT DOS Time to Unix:

```
// DOS: Packed format (year:7, month:4, day:5, hour:5, min:6, sec:5)
// Resolution: 2 seconds!
year = 1980 + (dos_date >> 9);
seconds = (dos_time & 0x1F) * 2;  // 2-second precision
```

ext2/3/4 Detection

All use same magic (0xEF53) - differentiate by feature flags:

```
ut32 feature_incompat = r_read_le32(&super.s_feature_incompat);
const char *fs_type = "ext2";

if (feature_incompat & 0x0004) {     // RECOVER flag
    fs_type = "ext3";                // Journaling support
}
if (feature_incompat & 0x0040) {     // EXTENTS flag
    fs_type = "ext4";                // Extent-based allocation
}
```

Metadata Extracted:

- Volume name, UUID, mount/write/check times
- Block/inode counts, sizes, free space
- Feature flags, filesystem state, creator OS

mn Command Demo

```
# ext3 filesystem
r2 test/bins/fs/ext3.img
mn

# NTFS filesystem
r2 test/bins/fs/ntfs.img
m /mnt ntfs 0
mn

# UBIFS filesystem
r2 test/bins/fs/ubifs.img
m /mnt ubifs 0
mn
```

Output: Formatted display with:

- Filesystem type and version
- Volume label, UUID/serial
- Timestamps (properly converted!)
- Geometry (block sizes, counts)
- [illegible]

rafs2 Tool

Standalone Filesystem Tool

Why?

- Leverage r2's filesystem plugin ecosystem
- One tool for all filesystem types
- No r2 session needed
- Scriptable operations

Supported Filesystems: ext2/3/4, FAT, NTFS, ISO9660, HFS, UBIFS, SSHFS

rafs2 Operations & Demo

Available Operations:

```
rafs2 -t [fstype] -l [path] [image]      # List directory
rafs2 -t [fstype] -c [path] [image]      # Read file contents
rafs2 -n [fstype] [image]                # List directory
```

Live Demo:

```
# List files in ext2 image
rafs2 -t ext2 -l /folder1 test/bins/fs/ext3.img
rafs2 -t ext2 -l /folder3 test/bins/fs/ext4.img

# Read file from UBIFS
rafs2 -t ubifs -c /etc/banner test/bins/fs/ubifs.img

# Get filesystem metadata
rafs2 -t ext2 -n test/bins/fs/ext3.img
```

Miknight Commander

Interactive TUI File Manager

Miknight Commander = Midnight Commander clone for r2

Features:

- Dual-panel file manager
- Works with `r_fs` mounts AND local filesystem
- Keyboard shortcuts + mouse support (because we can)
- Theme toggle (r2 palette vs classic MC)
- File operations: View, Copy, Delete, Mkdir
- Integrated viewer (text/hex)
- Interactive mode

Launch: `mmc`

Key Bindings:

Initially I wanted to keep the original Norton Commander keyboard shortcuts for some kind of historic precision. But discarded.

MMC Architecture

```c
typedef struct {
    char **entries;       // File/directory names
    char *types;          // Type flags ('d'=dir, 'f'=file, 'm'=mount)
    int count;            // Entry count
    int selected;         // Current selection
    int scroll_offset;    // Scroll position
    char *path;           // Current path
    bool is_fs_panel;     // r_fs vs local
} MMCPanel;

typedef struct {
    RCore *core;
    MMCPanel left, right;
    MMCPanel *active;
    bool use_r2_theme;
    MMCOrderMode ordering_mode;
} MMCState;
```

Design: Clean state management, no globals, dual-panel coordination

MMC Demo

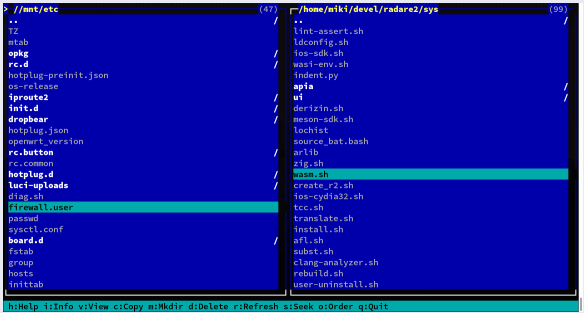


Figure 3: Miknight Commander in action

Live demonstration:

1. Mount UBIFS image
2. Navigate dual panels (r_fs mount + local filesystem)
3. View files with (text viewer with proper tab handling)
4. Copy files between panels with
5. Toggle theme with 'R' (r2 colors vs MC classic)
6. Mouse interactions (click to select, scroll wheel)

SSHFS Plugin

Remote Filesystem Access

Goal: Analyze/explore remote filesystems via SSH/SFTP from inside radare

- libssh2 integration for SSH/SFTP protocol
- Secure password input
- Multiple concurrent mounts supported
- Memory mapping support

URI Format: `/user:hostname[:port][:/remote/path]`

```
m /root:192.168.1.100 sshfs              # Basic
m /root:192.168.1.100:2222 sshfs         # Custom port
m /admin:server.com:22:/var/log sshfs    # Remote path
```

Why : not `@`? Avoids conflict with r2's `@` address operator!

SSHFS Memory Mapping

Challenge: Map remote files into r2's address space
Solution:

```
// From fs_sshfs.c - memory mapping approach:
char *uri = r_str_newf("malloc://%d", file->size);
RIODesc *desc = root->iob.open(root->iob.io, uri, R_PERM_RW, 0);

// Map at high address: 0x60000000 + (fd << 20)
ut64 map_addr = 0x60000000ULL + ((ut64)desc->fd << 20);
RIOMap *map = root->iob.map_add(root->iob.io, desc->fd,
                                R_PERM_RW, 0, map_addr, file->size);

// Read via SFTP, write to mapped region
libssh2_sftp_read(handle, buf, file->size);
root->iob.write_at(root->iob.io, file->off, buf, total);
```

SSHFS Demo

```
# Mount remote filesystem
r2 --
m /user:test-server:22 sshfs
# Password: [hidden input]

# List remote directory
md /user:test-server:22/etc
md /user:test-server:22/home

# Read remote file (memory-mapped!)
mc /user:test-server:22/etc/issue

# Use mis command on remote file
mis /user:test-server:22/etc/issue
f file 26 0x60400000

# Navigate using mmc and list functions of a binary
```

Use Case: Remote forensics without local file copies!

Design Patterns & Quality

All the filesystems share the same architecture. They can choose to implement features.

Consistent Architecture

Plugin System:

- Uniform `RFSPlugin` interface for all filesystems
- `.mount`, `.umount`, `.open`, `.read`, `.dir`, `.details` callbacks
- Optional features gracefully handled

Memory Management:

- No global state - all context in allocated structs
- `RFSRoot->ptr` stores per-mount context
- `RFSFile->ptr` stores per-file context
- Every `_mount` has corresponding `_umount` with cleanup

Error Handling:

- Early validation at function entry
- `goto fail` pattern for resource cleanup
- Boundary checks prevent buffer overflows

Cross-Platform Considerations

Endianness: All disk reads use `r_read_le32`/`r_read_le64`
Structure Packing: Direct buffer reads avoid compiler padding issues

```
// DON'T: ctx->leb_size = sb->leb_size;  (compiler-dependent!)
// DO:    ctx->leb_size = r_read_le32(sb_buf + 36);
```

lessons learned!

Use Cases

Ok, but any of this new stuff is useful, what can I do with it?

Applications

Firmware Security Analysis:

```
# Extract router firmware, analyze UBIFS
binwalk -e firmware.bin
r2 -qc 'm /mnt ubifs 0; md /mnt' test/bins/fs/ubifs.img
# Navigate, investigate config files, extract credentials, binaries...
```

Remote Forensics:

```
# Analyze compromised server remotely
r2 -qc 'm /forensics:suspect.com sshfs'
r2 -qc 'md /forensics:suspect.com/var/log'
r2 -qc 'mc /forensics:suspect.com/var/log/auth.log'
```

Embedded Device Analysis: IoT security research, CTF challenges, reverse engineering

Future Work and Planned Enhancements

UBIFS: LZO/ZSTD decompression inside r2

SSHFS: SSH key auth, write operations, connection pooling, timeouts

Adding New Filesystem Plugins:

- APFS (Apple File System)
- Btrfs (B-tree filesystem)
- F2FS (Flash-Friendly File System)
- JFFS2 (Journaling Flash File System v2)
- SquashFS

Tool Enhancements:

- rafs2: Write operations, batch mode
- mmc: Search, bookmarks, diff mode

Testing & Quality

To make our managers happy!

Test Coverage and sample files

- **UBIFS**: Mount/unmount, directory listing, file reading, decompression
- **mn command**: All 7 filesystems, UUID parsing, timestamp conversion
- **rafs2**: tests across several filesystem types
- Minimal filesystem images for each type
- Real-world samples (OpenWRT UBIFS)

CI Integration: Tests run on every commit, multiple platforms

Key Takeaways

Impact & Achievements

Enhanced Radare2 Capabilities:

- Good support for embedded filesystems (UBIFS)
- Comprehensive metadata extraction (7 filesystems by now)
- Remote analysis workflows enabled (SSHFS)
- radare2land-friendly tools (rafs2, mmc)

Benefits:

- Firmware analysis without extraction
- In-place filesystem examination
- Remote forensics capabilities
- Scriptable operations

For the Community:

- Extensible plugin architecture
- Reusable patterns and code

Questions & Contact

Let's Discuss!

Questions?
Since this is an online congress, reach me online. Any question, comment, suggestion, feature request is welcome.

- @mikelloc (X)
- @mikelloc@infosec.exchange (Mastodon)

(And if you're reading this outside the congress, this is just a PDF presentation, it cannot answer you, so better reach me online too)

EOF

Thank You!