Binary-Compatible Filesystem Verification with ACL2

Mihir Mehta, William R. Cook

Department of Computer Science
University of Texas at Austin
Austin, Texas 78703

mihir@cs.utexas.edu

09 September, 2019

Motivation

- Ubiquity of filesystems in application development, even when mediated through frameworks such as database systems
- Loss of stored data potentially catastrophic
- Crash consistency, concurrent operation, etc. now standard filesystem features, thus adding complexity
- With greater complexity, more important for filesystems to come with proofs

Binary compatibility

- Existing work (H. Sigurbjarnarson et al. 2016, H. Chen et al. 2017, A. Ileri et al. 2018) focused on the development of new filesystems with good properties
- Not always easy to install a new filesystem!
- Common scenario: an existing filesystem is known to be good, in terms of CPU usage/memory usage/fragmentation
- Need for techniques for verifying an existing filesystem, which may have filesystem-specific guarantees

Why FAT32?

- A non-trivially complex filesystem with features such as clustering and subdirectories
- Still widely used in embedded systems and removable media

Modeling FAT32

- Models of FAT32, called HiFAT and LoFAT, respectively model the filesystem at high and low levels of abstraction
- LoFAT designed as an ACL2 single-threaded object(*stobj*)
  - space-efficient in-memory representation of filesystem state
  - efficient array accesses and updates for the use of file operations
- HiFAT designed as a literal directory tree, matching an intuitive representation of the filesystem state
- Subset of POSIX file operations built up for LoFAT and proved correct with reference to their HiFAT implementations

FAT32 overview

FAT32 has three main on-disk data structures, which we replicate in LoFAT:

- The *data region*, split into `clusters` (pretty much like extents), which stores the contents of regular and directory files;
- The *file allocation table*, which stores `clusterchains` (linked lists), associated with files, which help reconstitute a given file's contents by pointing to all the clusters used by that file
- The *reserved area*, which store volume-level metadata about the size of a cluster, the number of clusters, and so on.

Designing file operations

- File table and file descriptor table implemented straightforwardly for syscalls such as `read` and `write`
- One goal of this model: facilitating correctness proofs for programs using the filesystem, which require *read-over-write* theorems to reason about sequences of file operations
- Tree representation used by HiFAT simplifies read-over-write proofs, but... what about LoFAT?
- Refinement, of course

How LoFAT refines HiFAT- I

- Transformation from LoFAT instances to HiFAT instances, `lofat-to-hifat`, defined to serve two purposes
  - It's a prerequisite for proving refinement.
  - Co-simulation raises the question: are two LoFAT instances equivalent even if not identical? Potential answer: maybe, if they transform to equivalent HiFAT instances

How LoFAT refines HiFAT- II

- Useful to define the inverse transformation, `hifat-to-lofat`
  - Read-only syscalls in HiFAT can be delegated from LoFAT: transform LoFAT to HiFAT then read
  - Other syscalls in HiFAT can be delegated from LoFAT: transform LoFAT to HiFAT then perform the syscall, then transform back
  - This leaves only a few syscalls, such as `statfs`, which need to be implemented directly in LoFAT in order to use volume-level metadata
- This gets us up and running with *verified* syscalls to build a co-simulation test suite

How LoFAT is made executable

- Layout of LoFAT designed to adhere to the layout of a `FAT32` disk image
- Therefore, possible to transform a LoFAT instance to a string (which can be written to a `FAT32` disk image) and back
- These transformations are verified to be inverses of each other, and this helps with co-simulation testing of programs which use the file operations provided by LoFAT

Co-simulation - I

- Testing LoFAT through co-simulation, to help make sure that it lines up with existing implementations of `FAT32`
- Run the same file operation on LoFAT and a canonical implementation - either the Linux `FAT32` implementation, or the `mtools` - and compare filesystem states after by transforming both and comparing the resulting HiFAT instances
  - This is done by an ACL2 program called `compare-disks`
  - The comparison is sound because of a proof that says the transformations between LoFAT and HiFAT are mutual inverses

Co-simulation - II

- For the co-simulation test suite, ACL2 programs written using LoFAT to replicate the operation of programs from the Coreutils and `mtools`
- Identical behaviour shown by the test suite, both in terms of standard output (compared by the `diff` program) and changes in filesystem state (compared by `compare-disks`)

Co-simulation test summary

Program
`cp`
`ls`
`mkdir`
`mv`
`rm`
`rmdir`
`stat`
`truncate`
`unlink`
`wc`

(a) Coreutils programs co-simulated

Program
`mcopy`
`mdel`
`mdeltree`
`mmd`
`mmove`
`mrd`
`mren`

(b) `mtools` programs co-simulated

Figure: Co-simulation tests

System calls implemented

Syscall	LoFAT implementation	LoFAT implementation through HiFAT transformation
`close`	✓	✓
`lstat`	✓	✓
`mkdir`		✓
`mknod`		✓
`open`	✓	✓
`pread`	✓	✓
`pwrite`		✓
`rename`		✓
`rmdir`		✓
`statfs`	✓	
`truncate`		✓
`unlink`		✓

Table: `POSIX` syscalls implemented

Equivalences

- `hifat-equiv`, an equivalence relation for HiFAT which disregards rearrangement of files and directories and access dates of files, defined for equivalential reasoning
- `lofat-equiv`, an equivalence relation for LoFAT, defined to apply to pairs of LoFAT instances which transform to the same HiFAT instance modulo `hifat-equiv`
- EqFAT, an equivalence relation for disk images (strings), defined to apply to those disk images which transform to the same LoFAT instance module `lofat-equiv`

Correctness of transformations

- Necessary for the transformation from LoFAT to HiFAT (`lofat-to-hifat`) to be the left inverse of the transformation from HiFAT to LoFAT (`hifat-to-lofat`), under an appropriate equivalence
- Choice of `hifat-equiv` for this purpose, even though it complicates the induction proof, to get an additional proof mostly for free
- `hifat-to-lofat` is the left inverse of `lofat-to-hifat` under `lofat-equiv`, because of the way the latter equivalence is defined

Optimizations

- Need to get the co-simulations to run in a reasonable amount of time, and also need to work with reasonably sized disk images (at least 1 GB)
- Use several ACL2 features, including stobjs and `mmap`-based read
- Further use ACL2's `mbe` (*must be equal*) to replace logically simple but slow sequences of operations with provably equivalent faster versions at the time of execution

Future Work

- Integration with FUSE to allow verification of programs written in other languages, not just ACL2
- Handling of new filesystems, with features such as journalling (crash consistency) and concurrent operation

Conclusion

- LoFAT constructed to model `FAT32` and provide a subset of the POSIX operations in a binary-compatible fashion
- HiFAT constructed to provably abstract LoFAT and help with the proof of read-over-write properties
- General technique for filesystem verification demonstrated