Model-Checking Support for File System Development

Wei Su, Yifei Liu, Gomathi Ganesan
Stony Brook University

Scott Smolka, Erez Zadok
Stony Brook University

Gerard Holzmann
Nimble Research

Geoff Kuenning
Harvey Mudd College

ABSTRACT
Developing and maintaining a file system is time-consuming, typically requiring years of effort. Developers often test compliance with APIs such as POSIX with hand-written regression suites that, alas, examine only a fraction of a file system’s state space. Conversely, formal model checking can explore vast state spaces efficiently, increasing confidence in the file system’s implementation. Yet model checking is not currently part of file system development. Our position is that file systems should be designed a priori to facilitate model checking. To this end, we introduce MCFS, an architecture for efficient and comprehensive file-system model checking. MCFS relies on two new APIs that save and restore a file system’s in-memory and on-disk state. We describe our earlier attempts at model-checking file systems, including unsuccessful or inefficient ones. Those attempts led us to develop VeriFS, which implements the new APIs. We illustrate MCFS’s model-checking principles with VeriFS, a FUSE-based file system we were able to quickly develop with MCFS’s help.

CCS CONCEPTS
• Software and its engineering → Software verification and validation; • Information systems → Information storage systems.

KEYWORDS
Model checking, file systems, verification

ACM Reference Format:

1 INTRODUCTION
File system development is time-consuming, complex, and error-prone, requiring precise logic, standards compliance (e.g., POSIX), and careful implementation of data structures and concurrency [14, 24, 49]. Yet even mature file systems suffer repeated bugs. For example, Btrfs [36] was designed 14 years ago yet reported 110 bugs in 2020 [21]. Such bugs can cause data corruption or loss and system crashes [22, 24, 37].

To help address this state of affairs, we present MCFS, a model-checking framework for file systems that: (1) checks every corner case in a bounded state space; (2) does not require an abstract file system model; (3) retains the original file system behavior; (4) applies to most file systems, whether in-kernel or user-space; and (5) has high performance.

MCFS compares file systems to each other with concurrent processes that non-deterministically issue file-system calls; it compares their outcomes (e.g., file content and return values) to find discrepancies. MCFS can exhaustively execute system-call permutations and thus uncover abnormal behavior.

We implemented MCFS using the Spin model checker [41] and applied it to a number of file systems. Our position is that thanks to its exhaustive state-space analysis capabilities, model checking should be an integral part of the file-system development process (alongside traditional hand-written regression suites [27, 39]). However, file systems need to facilitate model checking. We make the following contributions:

1. We created MCFS to support file system checking that can exhaustively search bounded state spaces.
2. We uncovered two inherent challenges to model-checking: cache incoherency and I/O inefficiencies. Ultimately we designed state checkpoint/restore APIs to ease integrating file systems with model checkers.
3. We developed (two versions of) a FUSE file system, VeriFS, that efficiently checkpoints and restores file-system states using our proposed APIs.
4. We empirically evaluated MCFS’s performance. Our results suggest that MCFS is a viable approach: one can use it to find behavioral deviations and bugs while developing or maintaining a file system.
Related work. Bug-detection techniques for file systems can be grouped into regression testing, verification, model checking, and fuzzing. Regression suites (e.g., xfstests [39] and ltp [27]) are useful in development and maintenance, but test only known defects and are unlikely to cover all corner cases. Prior work has demonstrated the benefits of building a machine-verifiable file system from scratch [2, 6–9, 20, 40, 50], but this approach does not apply to existing file systems. File-system model checkers have verified test cases [3, 28, 47, 48] and located corner-case bugs, but are strictly focused on crash consistency. Another approach is to build an abstract model and check whether a file system adheres to it [13, 23, 26, 34, 35]. Constructing a formally verifiable model, however, requires considerable domain knowledge and human effort. It is also impractical to build formal models for large and intricate file systems. Worse, the formal model requires updating when the file system’s code changes.

MCFS’s architecture has four main components: randomized test engines, optimized state-space exploration, integrity checks, and abstraction functions. Driven by the Spin model checker [15], randomized test engines nondeterministically issue operation sequences to each file system under consideration. Spin’s efficient partial-order reduction algorithm [19] allows MCFS to execute all permutations of the given set of calls and their parameters without duplication. Its randomized driver processes generate both valid and invalid call sequences. Valid sequences should succeed on all file systems, while invalid ones (e.g., write() before open()) should produce consistent error behavior. Invalid sequences are critical because they exercise error paths, where bugs often lurk.

We chose Spin to perform optimized state-space exploration because: (i) it is open-source and actively maintained; (ii) C code can be embedded in Spin’s model-description language (Promela) to issue system calls, and c_track statements let Spin record C buffers as states [16]; and (iii) Spin can perform parallel model checking using Swarm verification techniques [17], substantially speeding up exploration of large state spaces.

After each system call, integrity checks verify that all tested file systems have identical states by asserting equality of return values, error codes, file data, and metadata. If a discrepancy is detected, the integrity checker reports a potential bug and halts. Spin logs the precise sequence of operations, parameters, and starting and ending states that led to a problem, simplifying reproducibility. Because file systems have implementation-specific features [35], not all discrepancies are bugs. We believe though that many of these non-bug behavior differences are also interesting, since they still affect application behavior [33].

Finally, abstraction functions convert concrete states into abstract ones. In MCFS, each concrete state contains all the information needed to describe the file systems, including file data and metadata. MCFS uses the abstract state to determine whether a state was previously visited. If MCFS reaches a state that is logically equivalent to a previous one, it will backtrack corresponding concrete states to restore the file systems to their earlier versions. The abstraction functions hash the important data in the concrete states (including file paths, data, and relevant metadata) to distinguish logically unique states but omit noisy attributes such as time timestamps and the physical
3 CHALLENGES

We next explore a compromise in which Spin tracked only the persistent (on-disk) state would fail because of those less interesting attributes.

Designing abstraction functions requires domain knowledge. We wrote the functions to conform to the POSIX specification, so they are generic and applicable to most POSIX-compliant file systems.

3.1 Access to In-Memory States

MCFS must save and restore all information related to the tested file systems, including their persistent (on-disk) and dynamic (in-memory) states. Spin could use the underlying block device to track persistent states, but there is no simple way to access in-memory states—in kernel or user space—because they are not part of Spin itself.

In-kernel file systems. Many file systems (e.g., Ext4 [11], XFS [39] run in the Linux kernel, with states in the kernel address space. In theory, one could use /dev/kmem to save and restore those states, but in reality they are so intertwined with other kernel data structures that doing so is impractical.

User-space file systems. Tracking kernel file systems is hard, so we turned to file systems built on libFUSE [42] (e.g., fuse-ext2 [1]). Such file systems, however, are separate processes, so Spin cannot directly track their internal state. We tried several alternatives. First, we modified malloc to allocate memory from a shared-memory pool accessible to Spin, so that it could be saved and restored. This failed because important state was stored in static (non-heap) variables outside the shared-memory segment, leading to incorrect pointers and crashes. We concluded that it was impractical to modify file-system code to avoid these static variables.

3.2 Cache Incoherency

We next explored a compromise in which Spin tracked only the persistent (on-disk) state. Doing so allowed MCFS to run without crashing, but our experiments encountered corrupted file systems. A typical symptom was directory entries with corrupted or zeroed inodes, caused by Spin backtracking and restoring a persistent state. Since we were not restoring in-memory state to match, cached information in the kernel was no longer consistent with the disk content. For example, the dcache might contain a recently created directory, but the restored state might reflect a time before its creation.

We tried to resolve the inconsistency by calling fsync after each operation, and by mounting the file system with the sync option. Neither approach was effective: although they guaranteed that the caches were flushed to persistent storage, they did not implement the opposite operation—loading any Spin-initiated change in the persistent storage back into the in-memory caches.

Finally, we compromised again: we unmounted and remounted the file system between each pair of operations. An unmount is the only way to fully guarantee that no state remains in kernel memory. [Re]mounting always loads the latest state from disk, ensuring that the caches are coherent between each Spin state exploration. This compromise caused two problems: (1) it considerably slowed state exploration (see Section 6) and (2) it prevented us from identifying file-system bugs caused by incorrect in-memory states. These problems led us to consider adding support for file system state checkpointing and restoring (see Section 5).

3.3 State Explosion

Algorithm 1: Abstraction Functions

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Abstraction Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Path to the file system mount point path</td>
<td></td>
</tr>
<tr>
<td><strong>Output</strong>: 128-bit MD5 Hash</td>
<td></td>
</tr>
<tr>
<td>1. files ← list ([])</td>
<td></td>
</tr>
<tr>
<td>2. md5ctx ← md5_init ()</td>
<td></td>
</tr>
<tr>
<td>// Recursively walk the mount-point directory</td>
<td></td>
</tr>
<tr>
<td>3. foreach file in recursively_traverse_dir (path) do</td>
<td></td>
</tr>
<tr>
<td>4. files.append (file)</td>
<td></td>
</tr>
<tr>
<td>5. sort (files) // Sort files by pathnames</td>
<td></td>
</tr>
<tr>
<td>6. foreach file in files do</td>
<td></td>
</tr>
<tr>
<td>7. fd ← open (file.path)</td>
<td></td>
</tr>
<tr>
<td>8. content ← read (fd) // Read all file content</td>
<td></td>
</tr>
<tr>
<td>9. md5_update (md5ctx, content)</td>
<td></td>
</tr>
<tr>
<td>10. close (fd)</td>
<td></td>
</tr>
<tr>
<td>11. attrs ← stat (file)</td>
<td></td>
</tr>
<tr>
<td>// Get important metadata</td>
<td></td>
</tr>
<tr>
<td>12. attrs’ ← important_attributes (attrs)</td>
<td></td>
</tr>
<tr>
<td>13. md5_update (attrs’)</td>
<td></td>
</tr>
<tr>
<td>14. md5_update (file.path)</td>
<td></td>
</tr>
<tr>
<td>15. return get_md5_hash (md5ctx)</td>
<td></td>
</tr>
</tbody>
</table>

We use Spin’s c_track statement to declare memory buffers used by the C code. During state exploration, Spin detects an already-visited state by comparing these buffers against all previously visited states. This causes state explosion because any change in a buffer is considered a new state, yet some changes, such as access-time updates, are rarely relevant to bugs. Consequently, Spin could not fully explore file systems with even moderate parameter spaces. Fortunately, Spin’s c_track allows one to define abstract states used for matching, and concrete states used only in state restoration. We thus introduced an abstract state that contained an MD5 hash of file paths, data, and important metadata (e.g., mode, size, nlink, UID, and GID) for all files and directories.

Algorithm 1 shows the procedure to obtain an abstract state of a file system. We first identify all files and directories in the file system by traversing from the mount point. We
then sort them by their pathnames so that they appear in a consistent order. We then read each file's contents and call stat to obtain its metadata. The important attributes function extracts only the important metadata mentioned above. Finally, we calculate the MD5 hash for the files' content, important metadata, and pathnames.

We then marked persistent states as concrete, and instructed Spin to track only the hashes that distinguish different abstract states. Doing so not only prevented visiting duplicate states, it also greatly reduced the amount of memory needed to track states, increasing Spin's exploration capacity.

### 3.4 False Positives

MCFS performs integrity checks that assert state equality of the file systems under investigation. In case of any discrepancy, MCFS terminates and reports a bug, logging the operations it executed and their parameters. We found, however, that MCFS sometimes terminated on discrepancies that were not bugs. We took measures to prevent these false positives and describe several such cases below.

**Directory-size reporting and ordering.** In Ext4, directory sizes are a multiple of the block size; other file systems report sizes based on the number of active entries. Thus, directories on two different file systems might have the same contents but different sizes; we thus ignore directory sizes. Similarly, file systems return directory entries in different orders, so we sort the output of getdents before comparing them.

**Special folders.** Some file systems create special folders: For example, Ext4 has a lost+found folder to save lost and damaged files, while XFS does not. This caused namespace discrepancies between file systems. We added an exception list of special files and directories; MCFS ignores anything on this list when comparing abstract states.

**Differing data capacity.** Although we tested all file systems on block devices of the same size, they exposed different usable data capacities. This is a problem when the file systems are nearly full: calling write can succeed on one file system and fail on another, reporting a false bug. We thus equalize free space among file systems being checked: when MCFS starts, it queries all file systems and records the smallest free space $S_i$; then on each file system with free space $S_n$, it creates a dummy file and writes $S_n - S_i$ bytes of zeros.

None of these workarounds introduce false negatives, because they are all dealing with unstandardized behavior. For example, an application should not expect sorted output from getdents, so if a given file system suddenly stops sorting, that is not a bug. (However, if the change introduces other misbehavior, we will detect the consequences.)

### 4 MCFS Prototype Implementation

Figure 1 shows how MCFS's prototype handles different types of file systems. The file system syscall engine, written in Promela with embedded C code, consists of a multi-entry do ... od nondeterministic loop; each entry contains code to issue file-system operations, perform integrity checks, and record logs. Using Spin, MCFS nondeterministically selects an operation and its parameters, then executes it on all tested file systems. MCFS maps the file systems' backend storage devices into Spin's address space so it can track their states.

To prevent cache-coherency problems, MCFS unmounts and remounts kernel file systems (e.g., Ext4 and JFFS2) before and after each operation (see Section 3.2). However, syscalls such as write that depend on kernel state (e.g., open file descriptors) cannot be used in isolation. We thus developed meta-operations comprising small sequences of syscalls that avoid tracking kernel state: create_file creates and then closes a file; write_file opens, writes some data to, and closes a file. Other operations that can execute alone are run directly by MCFS, e.g., truncate and mkdir. Each operation's parameters are selected nondeterministically from a predefined (bounded) parameter pool. Because we limited our exploration space to fixed syscalls and parameters, the entire exploration—while large—is guaranteed to be bounded.

For FUSE-based file systems (e.g., fuse-ext2), the syscall is issued to the OS. FUSE's normal behavior results in several user/kernel messages being passed, coordinated via /dev/fuse (see fuse-ext2 in Figure 1).

To avoid being slowed by I/Os, we used RAM block devices as backend storage for block-based file systems (e.g., Ext4 and XFS). Linux's RAM block device driver (brd) requires all RAM disks to be the same size; we slightly modified it (renamed brd2), to allow different-sized RAM disks for file systems with different minimum-size requirements.

Some file systems must be mounted using special devices. For example, JFFS2 [44] requires an MTD character device [43] instead of a regular block device. MCFS sets up JFFS2 differently: it (1) loads the mtdram kernel module, which creates a virtual MTD device in RAM, and then (2) loads the mtdblock module to provide a block interface for the virtual MTD device. This approach allows Spin to mmap the MTD storage via the block device.

### 5 Tracking File-System States

Unmounting and remounting a file system after each operation solved cache-incoherency problems but slowed the model checking and deviated from a normal use case. Due to its backtracking search process, Spin saves and restores all information related to the tested file systems. Therefore, we must track all file-system states, including persistent and in-memory ones. We investigated three approaches: (1) process snapshotting and (2) VM snapshotting, which culminated in our (3) MCFS-enabled VeriFS.

**Process snapshotting.** User-space file systems run as independent processes. To keep their in-memory states
coherent with their persistent states, we can snapshot them using existing tools. We explored a popular snapshotting tool called CRIU [10] and tried integrating it with MCFS. Unfortunately, CRIU refused to checkpoint processes that have opened or mapped any character or block device (with a few unhelpful exceptions). Since FUSE-based file systems use the character device /dev/fuse to communicate with the kernel, CRIU could not checkpoint them. However, CRIU was able to snapshot the user-space NFS server Ganesha [32]; we are investigating model-checking Ganesha with CRIU.

Virtual-machine snapshotting. Hypervisors can snapshot and restore an entire VM, including full file-system states enclosed therein. However, VM-level snapshotting is fairly slow and heavyweight. For example, LightVM claims that it takes 30ms to checkpoint a trivial unikernel VM and 20ms to restore it [25]. This may be fast enough for cloud platforms but is too slow for MCFS; LightVM’s latency limited our model-checking rate to only 20–30 operations/s.

VeriFS. It is difficult for MCFS to track the in-memory state of file systems (see Section 3.1). But if a file system itself could checkpoint and restore its state, MCFS could use that facility to easily perform state capture and restoration, avoiding cache incoherence. To demonstrate this idea, we developed a RAM-based FUSE file system, VeriFS. Apart from standard POSIX operations such as open, write, and close, VeriFS provides checkpoint and restore APIs via ioctl1s: ioctl1_CHECKPOINT and ioctl1_RESTORE.

When MCFS calls ioctl1_CHECKPOINT with a 64-bit key, VeriFS locks itself, copies inode and file data into a snapshot pool under that key, and releases the lock.

Similarly, ioctl1_RESTORE causes VeriFS to query the snapshot pool for the given key. If it is found, VeriFS locks the file system, restores its full state, notifies the kernel to invalidate caches, unlocks the file system, and discards the snapshot.

VeriFS is intended to demonstrate the idea of having file systems themselves support model checking by providing checkpoint and restore APIs. Therefore, the initial version, VeriFS1, was fairly simple. It used a fixed-length inode array with a contiguous memory buffer attached to each inode as the file data. It had only a limited set of file system operations and lacked support for access(), rename(), symbolic and hard links, and extended attributes. It also did not limit the amount of data that could be stored.

We ran MCFS with Ext4 and VeriFS1 for over 5 days; MCFS executed over 159 million syscall operations without any errors, behavioral discrepancies, or file system corruption. To demonstrate how MCFS supports file system development, we next developed VeriFS2 to add missing features. During development, we used MCFS to model-check VeriFS1 against VeriFS2 to find and fix bugs in VeriFS2. MCFS helped us find several bugs in VeriFS2, which we discuss further in Section 6.

In sum, model checking a file system can involve exploring a vast number of states. If state exploration takes too long (e.g., is I/O-bound), then the entire model-checking process becomes impractical. Our position is that speedy and thorough file system model-checking requires the checkpoint/restore API we propose.

6 EVALUATION

We experimented with an MCFS prototype and a number of file systems running on a VM with 16 cores, 64GB RAM, and 128GB of swap space allocated on a local hypervisor SSD. We present preliminary performance results and discuss how MCFS helped our file system development.

Performance and memory demands. We ran MCFS and recorded key performance metrics for the following file system combinations: Ext2 vs. Ext4, Ext4 vs. XFS, Ext4 vs. JFFS2,Ext4 vs. XFS, Ext4 vs. JFFS2,Ext4 vs. XFS, Ext4 vs. JFFS2,Ext4 vs. XFFS2,Ext4 vs. JFFS2,Ext4 vs. JFFS2,Ext4 vs. JFFS2,Ext4 vs. JFFS2,Ext4 vs. JFFS2.
vs. JFFS2, and VeriFS1 vs. VeriFS2. We used 256KB RAM block devices for both Ext2 and Ext4, and 16MB for XFS, which allows a larger minimum file-system size. VeriFS is an in-memory file system and does not need a block device.

Figure 2 shows model-checking speeds observed in our experiments. Checking Ext4 vs. XFS on RAM disks was over 11× slower than Ext2 vs. Ext4 because MCFS consumed 105GB of swap space for the former, so swap time dominated. Checking Ext2 vs. Ext4 on HDD was 20× slower than on RAM disks; using SSD was still 18× slower. This illustrates the advantage of using RAM as backend storage. Checking VeriFS1 vs. VeriFS2 was 5.8× faster than Ext2 vs. Ext4 for two reasons: (i) MCFS used the checkpoint/restore APIs (see Section 5) and thus did not have to unmount and remount the VeriFS file systems; (ii) VeriFS runs entirely in-memory, so MCFS did not access block devices to checkpoint and restore persistent states.

Figure 3 shows MCFS’s speed when checking VeriFS1 over two weeks. MCFS maintained a rate of around 1,500 ops/s in the first 3 days; this rate then dropped drastically and swap usage spiked because Spin was resizing its hash table of visited states. After rebounding, MCFS’s speed gradually decreased over time because the checkpointed states could not fit in memory and it began to consume swap space. Its speed increased again between days 13 and 14 because the RAM hit rate was high (states needed by MCFS happened to be in memory and did not need to be swapped in and out).

Note that by default MCFS remounts and unmounts the file systems before and after each operation. To evaluate the impact of that approach, we also measured MCFS’s performance without the inter-operation remounts. The average speed for Ext2 vs. Ext4 (in RAM disks) was 316 ops/s, 38% faster than that when remounts and unmounts were used; and for Ext4 vs. XFS it was 34 ops/s, which is 70% faster.

Assisting file system development. While developing VeriFS1, we model-checked it vs. Ext4. MCFS found two bugs that we easily fixed, thanks to precise reports of operations and arguments. The first occurred after over 9K operations when test files on VeriFS and Ext4 had different data. VeriFS had failed to zero the file buffer if write created a hole in the file. The second bug was detected after over 1.2M operations: the test file in VeriFS2 was shorter than that in VeriFS1. The reason was that the write method in VeriFS2 updated the file size only when the file was expanded beyond its buffer capacity, rather than whenever the file was appended to.

7 CONCLUSIONS

We have proposed MCFS, a new model-checking framework for file system development that exhaustively explores a file system’s (bounded) state space, without requiring manual modeling or significant changes to the kernel or the file system itself. We developed VeriFS (v1 and v2), prototypes that demonstrate state checkpointing and restoration functionality via ioctl. These functions let MCFS track VeriFS’s full state while avoiding cache incoherence. MCFS found real bugs while we were developing VeriFS. Because both versions of VeriFS are simple and fast to model-check, they serve as a useful baseline against which we can compare other file systems.

We discussed the challenges we encountered when implementing MCFS, convincing ourselves of the need for file-system-level support to allow MCFS to work correctly and efficiently. Finally, while MCFS is designed to check file systems, its underlying approach is applicable to other system software.

Future work. We plan to add checkpoint/restore API support to Linux kernel file systems (e.g., Ext4) to eliminate the need for the current mount/remount workaround. We are implementing the checkpoint/restore API at the Linux VFS level, which we hope will apply to many Linux kernel file systems. We are also working on APIs that will checkpoint file system states to help us resume the model-checking process if an interruption occurs (e.g., due to a kernel crash). We also plan to run more than two file systems concurrently with MCFS and use a majority-voting approach to recognize incorrect file-system behavior.

We are exploring methods to track code coverage while model-checking. We will also use Spin’s swarm verification [17, 18] to explore larger state spaces in parallel.

8 ACKNOWLEDGMENTS

We thank the ACM HotStorage anonymous reviewers and our shepherd Ram Alagappan for their helpful feedback. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; and NSF awards CCF-1918225, CNS-1900706, CNS-1900589, CNS-1729939, CNS-1730726, DCL-2040599, and CPS-1446832.

REFERENCES


