# Using Model Checking to Find Serious File System Errors

Junfeng Yang, Paul Twohey, Dawson Engler *
{junfeng, twohey, engler}@cs.stanford.edu
*Computer Systems Laboratory*
*Stanford University*
*Stanford, CA 94305, U.S.A.*

Madanlal Musuvathi
madanm@microsoft.com
*Microsoft Research*
*One Microsoft Way*
*Redmond, WA 98052, U.S.A.*

## Abstract

This paper shows how to use model checking to find serious errors in file systems. Model checking is a formal verification technique tuned for finding corner-case errors by comprehensively exploring the state spaces defined by a system. File systems have two dynamics that make them attractive for such an approach. First, their errors are some of the most serious, since they can destroy persistent data and lead to unrecoverable corruption. Second, traditional testing needs an impractical, exponential number of test cases to check that the system will recover if it crashes at any point during execution. Model checking employs a variety of state-reducing techniques that allow it to explore such vast state spaces efficiently.

We built a system, FiSC, for model checking file systems. We applied it to three widely-used, heavily-tested file systems: ext3 [13], JFS [21], and ReiserFS [27]. We found serious bugs in all of them, 32 in total. Most have led to patches within a day of diagnosis. For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of metadata and entire directories, including the file system root directory "/".

## 1 Introduction

File system errors are some of the most destructive errors possible. Since almost all deployed file systems reside in the operating system kernel, even a simple error can crash the entire system, most likely in the midst of a mutation to stable state. Bugs in file system code can range from those that cause "mere" reboots to those that lead to unrecoverable errors in stable on disk state. In such cases, mindlessly rebooting the machine will not correct or mask the errors and, in fact, can make the situation worse.

*This research was supported by NSF grant CCR-0326227 and DARPA grant F29601-03-2-0117. Dawson Engler is partially supported by Coverity and an NSF Career award.

Not only are errors in file systems dangerous, file system code is simultaneously both difficult to reason about and difficult to test. The file system must correctly recover to an internally consistent state if the system crashes at *any* point, regardless of what data is being mutated, flushed or not flushed to disk, and what invariants have been violated. Anticipating all possible failures and correctly recovering from them is known to be hard; our results do not contradict this perception.

The importance of file system errors has led to the development of many file system stress test frameworks; two good ones are [24, 30]. However, these focus mostly on non-crash based errors such as checking that the file system operations create, delete and link objects correctly. Testing that a file system correctly recovers from a crash requires doing reconstruction and then comparing the reconstructed state to a known legal state. The cost of a single crash-reboot-reconstruct cycle (typically a minute or more) makes it impossible to test more than a tiny fraction of the exponential number of crash possibilities. Consequently, just when implementors need validation the most, testing is least effective. Thus, even heavily-tested systems have errors that only arise after they are deployed, making their errors all but impossible to eliminate or even replicate.

In this paper, we use model checking to systematically test and find errors in file systems. Model checking [5, 19, 22] is a formal verification technique that systematically enumerates the possible states of a system by exploring the nondeterministic events in the system. Model checkers employ various *state reduction* techniques to efficiently explore the resulting exponential state space. For instance, generated states can be stored in a hash table to avoid redundantly exploring the same state. Also, by inspecting the system state, model checkers can identify similar set of states and prioritize the search towards previously unexplored behaviors in the system. When applicable, such a systematic exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

The dominant cost of traditional model checking is the effort needed to write an abstract specification of the system (commonly referred to as the "model"). This up-front cost has traditionally made model checking completely impractical for large systems. A sufficiently detailed model can be as large as the checked system. Empirically, implementors often refuse to write them; those that are written have errors and, even if they do not, they "drift" as the implementation is modified but the model is not [6].

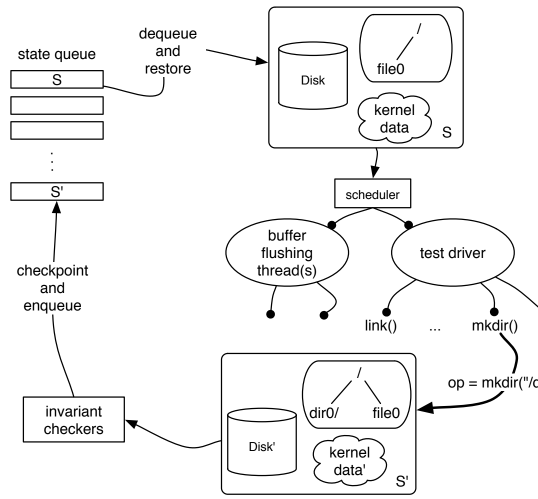
Recent work has developed *implementation-level* model checkers that check implementation code directly without requiring an abstract specification [18, 25, 26]. We leverage this approach to create a model checking infrastructure, the File System Checker (FiSC), which lets implementors model-check real, unmodified file systems with relatively little model checking knowledge. FiSC is built on CMC, an explicit state space, implementation model checker we developed in previous work [25, 26], which lets us run an entire operating system inside of the model checker. This allows us to check a file system *in situ* rather than attempting the difficult task of extracting it from the operating system kernel.

We applied FiSC to three widely-used, heavily-tested file systems, JFS [21], ReiserFS [27], and ext3 [13]. We found serious bugs in all of them, 32 in total. Most have led to patches within a day of diagnosis. For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of metadata and entire directories, including the file system root directory "`/`".

The rest of the paper is as follows. We give an overview of both FiSC (§2) and how to check a file system with it (§3). We then describe: the checks FiSC performs (§4), the optimizations it does (§5), and how it checks file system recovery code (§6). We then discuss results (§7) and our experiences using FiSC (§8), including sources of false positives and false negatives. We then conclude.

## 2 Checking Overview

Our system is comprised of four parts: (1) CMC, an explicit state model checker running the Linux kernel, (2) a file system test driver, (3) a permutation checker which verifies that a file system can recover no matter what order buffer cache contents are written to disk, and (4) a `fsck` recovery checker. The model checker starts in an initial pristine state (an empty, formatted disk) and recursively generates and checks successive states by systematically executing state transitions. Transitions are either test driver operations or FS-specific kernel threads which flush blocks to disk. The test driver is conceptually similar to a program run during testing. It creates, removes, and renames files, directories, and hard links; writes to and truncates files; and mounts and unmounts the file system. Figure 1 shows this process.




Model Checking Loop

```
while S = dequeue()
    for each op in S.operations
        S' =  checkpoint( op(restore(S) )
        check(S')
        if lookup(S'.disk, S'.abstract_fs) is new
            enqueue(S')
```

Figure 1: State exploration and checking overview. FiSC's main loop picks a state $S$ from the state queue and then iteratively generates its successor states by applying each possible operation to a restored copy of $S$. The generated state $S'$ is checked for validity and, if valid and not explored before, inserted onto the state queue.

As each new state is generated, we intercept all disk writes done by the checked file system and forward them to the permutation checker, which checks that the disk is in a state that `fsck` can repair to produce a valid file system after each subset of all possible disk writes. This avoids storing a separate state for each permutation and allows FiSC to choose which permutations to check. This checker is explained in Section 4.2. We run `fsck` on the *host* system outside of the model checker and use a small shared library to capture all the disk accesses `fsck` makes while repairing the file system generated by writing a permutation. We feed these `fsck` generated writes into the crash recovery checker. This checker allows FiSC to recursively check for failures in `fsck` and is covered in Section 6.

Figure 2 outlines the operation of the permutation and `fsck` recovery checkers. Both checkers copy the disk from the starting state of a transition and write onto the copy to avoid perturbing the system. After the copied disk is modified the model checker traverses its file system, recording the properties it checks for consistency in a model of the file system. Currently these are the name,

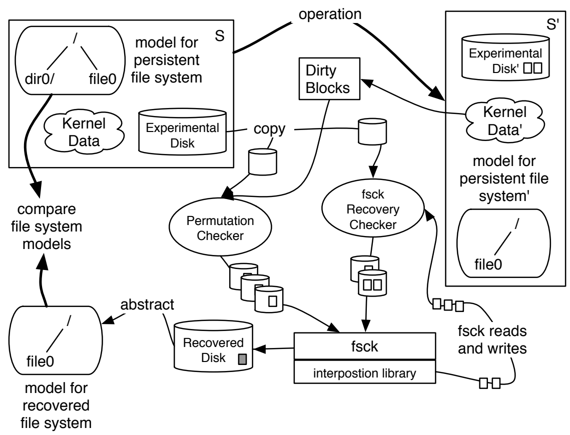



Figure 2: Disk permutation and `fsck` recovery checkers.

size, and link count of every file and directory in the system along with the contents of each directory. Note that this is a model of file system *data*, not file system *code*. The code to traverse, create, and manipulate the file system model mirrors the system call interface and can be reused to check many file systems. We check that this model matches one of the possible valid file systems, which are computed as in Section 3.2. An error is flagged in case of a mismatch.

After each new state is generated our system runs a series of invariant checkers looking for file system errors. If an error is found FiSC (1) emits an error message, (2) stores a trace for later error diagnosis that records all the nondeterministic choices it made to get to the error, and (3) discards the state. If there is no error, FiSC looks at the new state and puts it on the state queue if it has not already visited a similar state (§5.1). Otherwise, it discards the state.

New states are *checkpointed* and added to the state queue for later exploration. Checkpointing the kernel state captures the current execution environment so that it can be put on the state queue and *restored* later when the model checker decides to take it off the state queue and explore its operations. This state consists of the kernel heap and data, the disk, an abstract model of the current file system, and additional data necessary for invariant checks. As discussed in Section 5.1, FiSC searches the state space using breadth- or depth-first search along with some simple heuristics.

## 2.1 The Checking Environment

Similar to unit testing, model-checking a file system requires selecting two layers at which to cut the checked system. One layer defines the external interface that the test driver runs on. In our case we have the driver run atop the system call interface. The other layer provides a "fake environment" that the checked system runs on. We need this *environment model* because the checked file system does not run on bare hardware. Instead, FiSC provides a virtual block device that models a disk as a collection of sectors that can be written atomically. The block device driver layer is a natural place to cut as it is the only relatively well-documented boundary between in-core and persistent data.

Modern Unix derivatives provide a Virtual File System (VFS) interface [28]. While the VFS seems like a good place to cut, it varies significantly across operating systems and even across different versions of the same kernel. Further it has many functions with subtle dependencies. By instead cutting along the system call layer we avoid the headache of modeling these sparsely documented interactions. We also make our system easier to port and are able to check the VFS implementation, a design decision validated by the two bugs we found in the VFS code (§7).

# 3 Checking a New File System

This section gives an overview of what a file system implementor must do to check a new file system.

## 3.1 Basic Setup

Because CMC encases Linux, a file system that already runs within Linux and conforms to FiSC's assumptions of file system behavior will require relatively few modifications before it can be checked.

FiSC needs to know the minimum disk and memory sizes the file system requires. Ext3 had the smallest requirements: a 2MB disk and 16 pages of memory. ReiserFS had the highest: a 30MB disk and 128 pages. In addition, it needs the commands to make and recover the file system (usually with `mkfs` and `fsck` respectively). Ideally, the developer provides three different `fsck` options for: (1) default recovery, (2) "slow" full recovery, and (3) "fast" recovery that only replays the journal so that the three recovery modes may be checked against each other (§4).

In addition to providing these facts, an implementor may have to modify their file system to expose dirty blocks. Some consistency checks require knowing which buffers are dirty (§4.2). A file system, like ReiserFS, that uses its own machinery for tracking dirty buffers must be changed to explicitly indicate such dirty buffers.

When a file system fits within FiSC's model of how a file system works (as do ext3 and JFS) it takes a few days to start checking. On the other hand, ReiserFS took between one and two weeks of effort to run in FiSC as it violated one of the larger assumptions we made. As stated earlier, during crash checking FiSC mounts a copy of the disk used by the checked file system as a second block device that it uses to check the original. Thus, the file system must independently manage two disks in a reentrant manner. Unfortunately, ReiserFS does not do so: it uses a single kernel thread to perform journal writes for all mounted devices, which causes a deadlock when the journal thread writes to the log, FiSC suspends it, creates a copy of the disk, and then remounts the file system. Remounting normally replays the journal, but this requires writing to the journal – which deadlocks waiting for the suspended journal thread to run. We fixed the problem by modifying ReiserFS to not wake the journal thread when a clean file system is mounted read-only.

## 3.2 Modeling the File System

After every file system operation, FiSC compares the checked file system against what it believes is the correct *volatile file system* (VolatileFS). The VolatileFS reflects the effects of all file system operations done sequentially up through the last one. Because it is defined by various standards rather than being FS-specific, FiSC can construct it as follows. After FiSC performs an operation (e.g., `mkdir`, `link`) to the checked concrete system, it also emulates the operation's effect on a "fake" abstract file system. It then verifies that the checked and abstract file systems are equivalent using a lossy comparision that discards details such as time.

After every disk write, FiSC compares the checked file system against a model of what it believes to be the current *stable file system* (StableFS). The StableFS reflects the state the file system should recover to after a crash. At any point, running a file system's `fsck` repair utility on the current disk should always produce a file system equivalent to this StableFS.

Unlike the VolatileFS, the StableFS is FS-specific. Different file systems make wildly different guarantees as to what will be recovered after a crash. The ext2 [13] file system provides almost none, a journaling file system typically intends to recover up to the last completed log record or commit point, and a soft-updates [17] file system recovers to a difficult-to-specify mix of old and new data.

Determining how the StableFS evolves requires determining two FS-specific facts: (1) when it can legally change and (2) what it changes to. FiSC requires that the implementor modify the checked file system to call into the model checker to indicate when the StableFS changes. For journaling file systems this change typically occurs when a journal commit record is written to disk. We were able to identify and annotate the commit records relatively easily for ext3 and ReiserFS. JFS was more difficult. In the end, after a variety of false starts, we gave up trying to determine which journal write represented a commit-point and instead let the StableFS change after *any* journal write. We assume a file system implementor could do a better job.

Once we know that the StableFS changes, we need to know what it changes to. Doing so is difficult since it essentially requires writing a crash recovery specification for each file system. While we assume a file system implementor could do so, we check systems we did not build. Thus, we take a shortcut and use `fsck` to generate the StableFS for us. We copy the experimental disk, run `fsck` to reconstruct a file system image after the committed operations, and traverse the file system, recording properties of interest. This approach can miss errors since we have no guarantee that `fsck` will produce the correct state. However, it is relatively unlikely that `fsck` will fail when repairing a perfectly behaving disk. It is even more unlikely that if it does fail that it will do so in the same way for the many subsequent crashed disks to which the persistent file system model will be compared.

## 3.3 Checking More Thoroughly

Once a basic file system is up and being checked, there are three main strategies an implementor can follow to check their file system more thoroughly: downscaling [10], canonicalization, and exposing choice points. We talk about each below.

**Downscale**. Operationally this means making everything as small as plausible. Caches become one or two entries large, file systems just a few "nodes" (where a node is a file or directory). Model checking works best at ferreting out complex interactions of a small number of nouns (files, directories, blocks, threads, etc) since this small number allows caching techniques to give the most leverage. There were three main places we downscaled. First, making disk small (megabytes rather than gigabytes). Second, checking small file system topologies, typically 2-4 nodes. Finally, reducing the size of "virtual memory" of the checked Linux system to a small number of pages.

**Canonicalization**. This technique modifies states so that state hashing will not see "irrelevant" differences. In practice, the most common canonicalization is to set as many things as possible to constant values: clearing inode generation numbers, mount counts, time fields; zeroing freed memory and unused disk blocks (especially journal blocks).

Many canonicalizations require FS-specific knowledge and thus must be done by the implementor. However, FiSC does do two generic canonicalization. First, it constrains the search space by only writing two different values to data blocks, significantly reducing the number of states while still providing enough resolution to catch data errors. Second, before hashing a model of a file system, FiSC transforms the file system to remove superficial differences, by renaming files and directories so that there is always a sequential numbering among file system objects. For example a file system with one directory and three files "a," "b," and "c" will have the same model as another file system with one directory and three files "1," "2," and "3" if the files have the same length and content. Canonicalization lets us move our search space away from searching for rare filename-specific bugs and toward the relatively more common bugs that arise while creating many file system topologies.

**Expose choice points**. Making sources of nondeterminism ("choice points") visible to FiSC lets it search the set of possible file system behaviors more thoroughly. A low-level example is adding code to fail FS-specific allocators. More generally, whenever a file system makes a decision based on an arbitrary time constraint or environmental feature, we change it to call into FiSC so that FiSC can choose to explore each possible decision in every state that reaches that point.

Mechanically, exposing a choice point reduces to modifying the file system code to call "`choose(`$n$`)`" where $n$ is the number of possible decision alternatives. `choose` will appear to return to this callsite $n$ times, with the return values $0, \ldots, (n-1)$. The caller uses this return value to pick which of the $n$ possible actions to perform. An example: both ReiserFS and ext3 flush their in-memory journals to disk after a given amount of time has lapsed. We replaced this time check with a call to `choose(2)` and modified the caller so that when `choose` returns 0 the code flushes the commit record; when it returns 1 it does not. As another example, file systems check the buffer cache before issuing disk reads. Without care, this means that the "cache miss" path will rarely be checked (especially since we check tiny file system topologies). We solve this problem by using `choose` on the success path of the buffer cache read routine to ensure FiSC also explores the miss path. In addition, FiSC generically fails memory allocation routines and permission checks.

When inserting choice points, the implementor can exploit well-defined internal interfaces to increase the set of explored actions. Interface specifications typically allow a range of actions, of which an implementation will pick some subset. For example, many routines specify that any invocation may return an "out of memory" error. However their actual implementation may only allocate memory on certain paths, or perhaps never do any allocations at all. It is a mistake to only fail the specific allocation calls an implementation performs since this almost certainly means that many callers and system configurations will never see such failures. The simple fix is to insert a choice point as the routine's first action allowing the model checker to test that failure is handled on each call.

Unfortunately, it is not always easy to expose choice points and may require restructuring parts of the system to remove artificial constraints. The most invasive example of these modifications are the changes to the buffer cache we made so that the permutation checker (§4.2) would be able to see all possible buffer write orderings.

# 4 Checkers

This section describes the checks FiSC performs.

## 4.1 Generic Checks

FiSC inspects the actual state of the system and can thus catch errors that are difficult or impossible to diagnose with static analysis. It is capable of doing a set of general checks that could apply to any code run in the kernel:

**Deadlock**. We instrument the lock acquisition and release routines to check for circular waits.

**NULL**. FiSC reports an error whenever the kernel dereferences a NULL pointer.

**Paired functions**. There are some kernel functions, like `iget`, `iput` for inode allocation and `dget`, `dput` for directory cache entries, which should always be called in pairs. We instrument these functions in the kernel and then check that they are always called in pairs while running the model checker.

**Memory leak**. We instrument the memory allocation and deallocation functions so FiSC can track currently used memory. We also altered the system-wide freelist to prevent memory consumers from allocating objects without the model checker's knowledge. After every state transition we stop the system and perform a conservative traversal [2] of the stack and the heap looking for allocated memory with no references.

**No silent failures**. The kernel does not request a resource for which it does not have a specific use planned. Thus, it is likely a bug when a system call returns success after it calls a resource allocation routine that fails. The exception to this pattern is when code loops until it acquires a resource. In which case, we generate a false positive when a function fails during the first iteration of the loop but later succeeds. We suppress these false positives by manually marking functions with resource acquisition loops.

## 4.2 Consistency Checks

FiSC checks the following consistency properties.

**System calls map to actions**. A mutation of the file system that indicates success (usually a system call with a return value of zero) should produce a user-visible change, while an indication of failure should produce no such change. We use a reference model (the VolatileFS) to ensure that when an operation produces a user-visible change it is the correct change.

**Recoverable disk write ordering**. A described in §2, we write arbitrary combinations of dirty buffer cache entries to disk, checking that the system recovers to a valid state. File system recovery code typically requires that disk writes happen in certain stylized orders. Illegal orders may not interfere with normal system operation, but will lead to unrecoverable data loss if a crash occurs at an inopportune moment. Comprehensively checking for these errors requires we (1) have the largest legal set of possible dirty buffers in memory and (2) flush combinations of these blocks to disk at every legal opportunity. Unfortunately, many file systems (all those we check) thwart these desires by using a background thread to periodically write dirty blocks to disk. These cleaned blocks will not be available for subsequent reorder checking, falsely constraining the schedules we can generate. Further, the vagaries of thread scheduling can hide vulnerabilities — if the thread does not run when the system is in a vulnerable state then the dangerous disk writes will not happen. Thus we modified this thread to do nothing and instead have the model checker track all blocks that could be legally written. Whenever a block is added to this set we write out different permutations of the set, and verify that running `fsck` produces a valid file system image. The set of possible blocks that can be written are (1) all dirty buffers in the buffer cache (dirty buffers may be written in any order) and (2) all requests in the disk queue (disks routinely reorder the disk queue).

This set is initially empty. Blocks are added whenever a buffer cache entry is marked dirty. Blocks are removed from this set in four ways: (1) they are deleted from the buffer cache, (2) marked clean, (3) the file system explicitly waits for the block to be written or (4) the file system forces a synchronous write of a specific buffer or the entire disk request queue.

**Changed buffers are marked dirty**. When a file system changes a block in the buffer cache it needs to mark it as dirty so the operating system knows it should eventually write the block back to disk. Blocks that are not marked as dirty may be flushed from the cache at any time. Initially we thought we could use the generic dirty bit associated with each buffer to track the "dirtiness" of a buffer, but each file system has a slightly different concept of what it means for a buffer to be dirty. For example, ext3 considers a buffer dirty if one of the following conditions is true: (1) the generic dirty bit is set, (2) the buffer is journaled and the journal dirty bit is set, or (3) the buffer is journaled and it has been revoked and the revocation is valid. Discovering dirty buffer invariants requires intimate knowledge of the file system design; thus we have only run this checker on ext3.

**Buffer consistency**. Each journaling file system associates state with each buffer it uses from the buffer cache and has rules about how that state may change. For example a buffer managed by ext3 may not be marked both dirty and "journal dirty." That is, it should be written first to the journal (journal dirty), and then written to the appropriate location on disk (dirty).

**Double fsck**. By default `fsck` on a journaled file system simply replays the journal. We compare the file system resulting from recovering in this manner with one generated after running `fsck` in a comprehensive mode that scans the entire disk checking for consistency. If they differ, at least one is wrong.

## 5 Scaling the System

As we brought our system online we ran into a number of performance and memory bottlenecks. This section describes our most important optimizations.

### 5.1 State Hashing and Search

Exploring an exponential state space is a game where you ignore (hopefully) irrelevant details in a quest to only explore states that differ in non-superficial ways. FiSC plays this game in two places: (1) state hashing, where it selectively discards details to make bit-level different states equivalent and (2) searching, when it picks the next state to explore. We describe both below.

We initially hashed most things in the checked file system's state, such as the heap, data segment, and the raw disk. In practice this meant it was hard to comprehensively explore "interesting" states since the model checker spent its time re-exploring states that that were not that much different from each other. After iterative experimentation we settled on only hashing the VolatileFS, the StableFS, and the list of currently runnable threads. We ignore the heap, thread stacks, and data segment. Users can optionally hash the actual disk image instead of the more abstract StableFS to check at a higher-level of detail.

Despite discarding so much detail we rarely can explore all states. Given the size of each checkpoint (roughly 1-3MB), the state queue holding all "to-be-explored" states consumes all memory long before FiSC can exhaust the search space. We stave this exhaustion off by randomly discarding states from the state queue whenever its size exceeds a user-selected threshold.

We provide two heuristic search strategies as alternatives to vanilla DFS or BFS. The first heuristic attempts to stress a file system's recovery code by preferentially running states whose disks will likely take the most work to repair after a crash. It crudely does so by tracking how many sectors were written when the state's parent's disk was recovered and sorts states accordingly. This approach found a data loss error in JFS that we have not been able to trigger with any other strategy.

The second heuristic tries to quantify how different a given state is from previously explored states using a utility score. A state's utility score is based on how many times states with the same features have already been explored. Features include: the number of dirty blocks a state has, its abstract file system topology, and whether

| | ext3 | ReiserFS | JFS |
|---|---|---|---|
| **States** | | | |
| Total | 10800 | 630 | 4500 |
| Expanded States | 2419 | 142 | 905 |
| State Transitions | 35978 | 11009 | 14387 |
| **Time** | | | |
| with Memoization | 650 | 893 | 3774 |
| without Memoization | 7307 | 29419 | 4343 |

Table 1: The number of states, transitions, and the cost of checking each file system until the point at which FiSC runs out of memory. Times are all in seconds. ReiserFS's relatively large virtual memory requirements limited FiSC checks to roughly an order of magnitude fewer states than the other systems. `fsck` memoization (described in §5.4) speeds checking of ext3 by a factor of 10, and ReiserFS by a factor of 33.

its parent executed new file system statements. A state's score is an exponentially-weighted weighted sum of the number of times each feature has been seen.

## 5.2 Systematically Failing Functions

When a transition (e.g., `mkdir`, `creat`) is executed, it may perform many different calls to functions that can fail such as memory allocation or permission checks (§3.3). Blindly causing all combinations of these functions to fail risks having FiSC explore an exponential number of uninteresting, redundant transitions for each state. Additionally, in many cases FS-implementors are relatively uninterested in "unlikely" failures, for example, those only triggered when both memory allocation failures and a disk read error occurs.

Instead, we use an iterative approach — FiSC will first run a transition with no failures, it will then run it failing only a single callsite until all callsites have been failed, it will then similarly fail two callsites, etc. Users can specify the maximum number of failures that FiSC will explore up to. The default is one failure. This approach will find the smallest possible number of failures needed to trigger an error.

## 5.3 Efficiently Modeling Large Disks

As Figure 3 shows, naively modeling reasonable-sized disks with one contiguous memory allocation severely limits the number of states our model checker can explore as we quickly exhaust available memory. Changing file system code so that it works with a smaller disk is non-trivial and error prone as the code contains mutually-dependent macros, structures, and functions that all rely on offsets in intricate ways. Instead we efficiently model large disks using hash compaction [31]. We keep a database of disk *chunks*, collections of disk sectors, and their hashes. The disk is thus an array of references to hashed chunks. When a write alters a chunk we hash the new value, inserting it into the database if necessary, and have the chunk reference the hash.

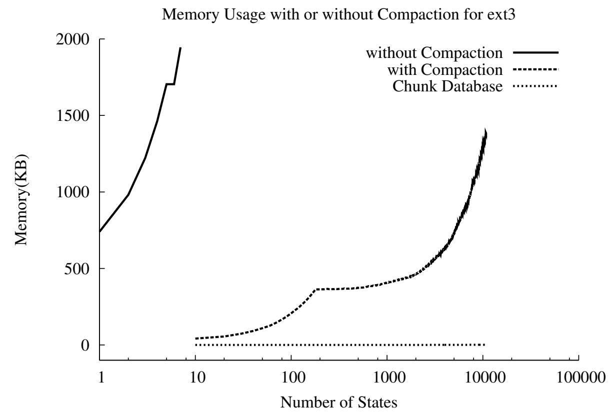



Figure 3: Memory usage when model checking ext3 on a 40MB disk with and without disk compaction. Without compaction the model checker quickly exhausts all the physical memory and dies before it reaches 20 states. The chunk database consumes about 0.2% of the total memory with a maximum of less than 2MB. Note, the spike around the 150 state mark happens because FiSC starts randomly discarding states from its state queue.

## 5.4 fsck Memoization

Repairing a file system is expensive. It takes about five times as long to run `fsck` as it does to restore a state and generate a new state by executing an operation. If we are not careful, the time to run `fsck` dominates checking. Fortunately, for all practical purposes, recovery code is deterministic: given the same input disk image it should always produce the same output image. This determinism allows us to memoize the result of recovering a specific disk. Before running `fsck` we check if the current disk is in a hash table and, if so, return the already computed result. Otherwise we run `fsck` and add the entry to the table. (As a space optimization we actually just track the sectors read and written by `fsck`.) While memoization is trivial, it gives a huge performance win as seen in Table 1, especially since our `fsck` recovery checker (§6) can run `fsck` 10-20 times after each crash.

## 5.5 Cluster-based Model Checking

A model checking run makes a set of configuration choices: the number of files and directories to allow, what operations can fail, whether crashes happen during recovery, etc. Exploring different values is expensive, but not doing so can miss bugs. Fortunately, exploration is easily parallelizable. We wrote a script that given a set of configuration settings and remote machines, generates all configurations and remotely executes them.

## 5.6 Summary

Table 1 shows that FiSC was able to check more than 10k states and more than 35k transitions for ext3 within 650 seconds. The expanded states are those for which all their possible transitions are explored. The data in this section was computed using a Pentium 4 3.2GHz machine with 2GB memory.

# 6 Crashes During Recovery

A classic recovery mistake is to incorrectly handle a crash that happens during recovery. The number of potential failure scenarios caused by one failure is unwieldy, the number of scenarios caused by a second failure is combinatorially exciting. Unfortunately, since many failures are correlated, such crashes are not uncommon. For example, after a power outage, one might run `fsck` only to have the power go out again while it runs. Similarly, a bad memory board will cause a system crash and then promptly cause another one during recovery.

This section describes how we check that a file system's recovery logic can handle a single crash during recovery. We check that if `fsck` crashes during its first recovery attempt, the final file system (the StableFS) obtained after running `fsck` a second time (on a disk possibly already modified by the previous run) should be the same as if the first attempt succeeded. We do not consider the case where `fsck` crashes repeatedly during recovery. While repeated failure is intellectually interesting, the difficulty in reasoning about errors caused by a single crash is such that implementors have shown a marked disinterest in more elaborate combinations.

Conceptually, the basic algorithm is simple:

1. Given the disk image $d_0$ after a crash, run `fsck` to completion. We record an ordered "write-list" $WS = (w_1, \ldots, w_n)$ of the sectors and values written by `fsck` during recovery. Here $w_i$ is a tuple $\langle s_i, v_i \rangle$, where $s_i$ is the sector written to and $v_i$ is the data written to the sector. In more formal terms, we model `fsck` as a function (denoted *fsck*) that maps from an input disk $d$ to an output disk $d'$, where the differences between $d$ and $d'$ are the values in the write-set $WS$. For our purposes these writes are the only effects that running `fsck` has. Moreover, we denote the partial evaluation of $fsck(d)$ after performing writes $w_1, \ldots, w_i$ as $fsck_{[i]}(d)$. By definition, $fsck(d) \equiv fsck_{[n]}(d)$.
2. Let $d_i$ be the disk image obtained by applying the writes $w_1, \ldots, w_i$ to disk image $d_0$. This is the disk image returned by $fsck_{[i]}(d_0)$. Next, rerun `fsck` on $d_i$ to verify that it produces the same file system as running it on $d_0$ (i.e., $fsck(d_i) = fsck(d_0)$). Computing $fsck(d_i) \equiv fsck(fsck_{[i]}(d_0))$ simulates the effect of a crash during the recovery where `fsck` performed $i$ writes and then was restarted.

To illustrate, if invoking $fsck(d_0)$ writes two sectors 1 and then 4 with values $v_1$, and $v_2$ respectively, the algorithm will first apply the write $\langle 1, v_1 \rangle$ to $d_0$ to obtain $d_1$, crash, check, and then apply write $\langle 4, v_2 \rangle$ to $d_1$ to obtain $d_2$, crash and check.

This approach requires three refinements before it is reasonable. The first is for speed, the second to catch more errors, and the third to reason about them. We describe all three below.

## 6.1 Speed From Determinism

The naive algorithm checks many more cases than it needs to. We can dramatically reduce this number by exploiting two facts. First, for all practical purposes we can regard `fsck` as a deterministic procedure (§5.4). Determinism implies a useful property: if two invocations of a deterministic function read the same input values, then they must compute the same result. Thus, if a given write by `fsck` does *not change* any value it previously read then there is no need to crash and rerun it — it will always get back to the current state. Second, `fsck` rarely writes data it reads. As a result, most writes do not require that we crash and recover: they will not intersect the set of sectors `fsck` reads and thus, by determinism, cannot influence the disk it would produce.

We state this independence more precisely as follows. Let $RS_i = \{r_1, \ldots, r_k\}$ denote the (unordered) set of all sectors read by $fsck_{[i]}(d)$. As above, let $d_i$ denote the disk produced by applying the writes $(w_1, \ldots, w_i)$ in order to the initial disk $d_0$. We claim that if the sector $s_k$ written by $w_k$ is not in the read set $RS_i$, then running *fsck* to completion on disk $d_i$ produces the same result as running it on $d_{i-1}$. I.e., $s_i \notin RS_i$ implies $fsck(d_i) = fsck(d_{i-1})$ (recall that $RS_{i-1} \subseteq RS_i$). Tautologically, a deterministic function can only change what it computes if the values it reads are different. Thus, $s_i \notin RS_i$ implies that $fsck(d_i)$ and $fsck(d_{i-1})$ read identical values for the first $i - 1$ steps, forcing them to both compute the same results so far. Then, at step $i$, both will perform write $w_i$, making their disks identical.

There are two special cases that our checker exploits to skip running `fsck`:

1. Suppose $w_i$ does write a sector in $RS_i$, but the value it writes is the same as what is currently on disk (i.e., $d_i = d_{i-1}$). Clearly if $d_{i-1} = d_i$ then $fsck(d_i) = fsck(d_{i-1})$. Surprisingly, recovery programs empirically do many such redundant writes.
2. If (1) the sector $s_i$ written by $w_i$ is dominated by an earlier write $w_j$ and (2) there is no read that precedes $w_j$, then $w_i$ cannot have any affect since $s_i$ will always be overwritten with $v_j$ when `fsck` is restarted.

## 6.2 Checking All Write Orderings

As described the algorithm can miss many errors. Sector writes can complete in any order unless (1) they are explicitly done synchronously (e.g., using the O_DIRECT option on Unix) or (2) they are blocked by a "sync barrier," such as the sync system call on Unix which (is believed to) only return when all dirty blocks have been written to disk. Thus, generating all disk images possible after crashing fsck at any point requires partitioning the writes into "sync groups" and checking that fsck would work correctly if it was rerun after performing any subset of the writes in a sync group (i.e., the power set of the writes contained in the sync group). For example, if we write sectors 1 and 2, call sync, then write sector 4, we will have two sync groups $S_0 = \{1, 2\}$ and $S_1 = \{4\}$. The first, $S_0$, generates three different write schedules: $\{(1), (2), (1, 2)\}$. A write schedule is the sectors that were written before fsck crashed (note that the schedule $(2, 1)$ is equivalent to $(1, 2)$ since we rerun fsck after both writes complete). Given a sync group $S_i$ our checker does one of two things.

1. If the size of $S_i$ is less than or equal to a user-defined threshold, $t$, the checker will exhaustively verify all different interleavings.
2. If the size is larger than $t$ the checker will do a series of trials, where it picks a random subset of random size within $S_i$. These trials can be deterministically replayed later because we seed the pseudo-random number generator with a hash of the sectors involved.

We typically set $t = 5$ and the number of random trials to 7. Without this reordering we found no bugs in fsck recovery; with it we have found them in all three checked file systems.

## 6.3 Finding the Right Perspective

Unfortunately, while recovery errors are important, reasoning about them is extremely difficult. For most recovery errors the information the model checker provides is of the form "you wrote block 17 and block 38 and now your disk image has no files below '/'." Figuring out (1) semantically what was being done when this error occurred, (2) what the blocks are for, and (3) why writing these values caused the problem can easily take an entire day. Further, this process has to be replicated by the implementor who must fix it. Thus, we want to find the simplest possible error case. The checker has five modes, described below and roughly ordered in increasing degrees of freedom and hence difficulty in diagnosing errors. (Limiting degrees of freedom also means they are ordered by increasing cost.) At first blush, five modes might seem excessive. In reality they are a somewhat desperate attempt to find a perspective that makes reasoning about an error tractable. If we could think of additional useful views we would add them.

**Synchronous, atomic, logical writes.** The first, simplest view is to group all sector writes into "logical writes" and do these synchronously (i.e., in the order that they occur in program execution). Here, logical writes means we group all blocks written by the same system call invocation as one group. I.e., if there are two calls to the write system call, the first writing sectors $l_0 = (1, 2, 3)$ and the second writing sectors $l_1 = (7, 8)$ we have two logical operations, $l_0$ and $l_1$. We apply all the writes in $l_0$, crash, check, apply the writes in $l_1$ crash, and check.

This is the strongest logical view one can have of disk: all operations complete in the order they were issued and all the writes in a single logical operation occur atomically. It is relatively easy to reason about these errors since it just means that fsck was not reentrant.

**Synchronous, non-atomic, left-to-right logical writes.** Here we still treat logical writes as synchronous, but write their contained sectors non-atomically, left-to-right. I.e., write the first sector in a logical group, crash, check, then write the next, etc. These errors are also relatively easy to reason about and tend to be localized to a single invocation of a system call where a data structure that was assumed to be internally consistent straddled a sector boundary.

**Reordered, atomic logical writes.** This mode reorders all logical writes within the same sync group, checking each permutation. These errors can often be fixed by inserting a single sync call.

**Synchronous, non-atomic logical writes.** This mode writes the sectors within a logical operation in any order, crashing after each possible schedule. These errors are modular, but can have interesting effects.

**Reordered sector writes.** This view is the hardest to reason about, but the sternest test of file system recovery: reorder all sectors within a sync group arbitrarily. We do not like these errors, and if we hit them will make every attempt to find them with one of the prior modes.

# 7 Results

Table 2 summarizes the errors we found, broken down by file systems and categories. All errors were reported to the respective developers. We found 32 serious bugs in total; 21 have been fixed and 9 of the remaining 11 confirmed. The latter were complex enough that no patch has been submitted. There were 10 errors where supposedly stable, committed data and metadata (typically entire directories) could be lost. JFS has higher error counts in part due to the immediate responses from the JFS developers, which enabled us to patch the errors and continue checking JFS. We are currently checking the XFS file system and the preliminary results are promising. [1]

We discuss the bug categories in more detail below, highlighting the more interesting errors found.

[1] The errors reported in this paper can be found at page http://keeda.stanford.edu/~junfeng/osdi-fisc-bugs.html, titled "OSDI FiSC Bugs."

| Error type | VFS | ext2 | ext3 | JFS | ReiserFS | total |
|---|---|---|---|---|---|---|
| Lost stable data | n/a | n/a | 1 | 8 | 1 | 10 |
| False clean | n/a | n/a | 1 | 1 | | 2 |
| Security holes | | 2 | 2 (minor) | 1 | | 5 |
| Kernel crashes | 1 | | | 10 | 1 | 12 |
| Other (serious) | 1 | | 1 | 1 | | 3 |
| Total | 2 | 2 | 5 | 21 | 2 | 32 |

Table 2: We found 32 errors, 10 of which could cause permanent data loss. There are 3 intended errors, where programmers decided to sacrifice consistency for availability. They are not shown in this table.

## 7.1 Unrecoverable Data Loss

The most serious errors we found caused the irrevocable loss of committed, stable data. There were 10 such errors where an execution sequence would lead to the complete loss of metadata (and its associated data) that was committed to the on-disk journal. In several cases, all or large parts of long-lived directories, including the root directory "`/`", were obliterated. Data loss had two main causes: (1) invalid write ordering of the journal and data during recovery and (2) buggy implementations of transaction abort and `fsck`.

**Invalid recovery write ordering**. There were three bugs of this type. During normal operation of a journaling file system the journal must be flushed to disk before the data it describes. The file systems we check seem to get this right. However, they all get the inverse of this ordering constraint wrong: during recovery, when the journal is being replayed, all data modified by this roll forward must be flushed to disk before the journal is persistently cleared. Otherwise, if a crash occurs, the file system will be corrupt or missing data, but the journal will be empty and hence unable to repair the file system.

Figure 4 gives a representative error from the ext3 `fsck` program. The chain of mishaps is as follows:

1. `recover_ext3_journal` rolls the journal forward by calling `journal_recover`.
2. `journal_recover` replays the journal, writing to the file system using cached writes. It then calls `fsync_no_super` to flush all the modified data back to disk. However, this macro has been defined to do nothing due to an error made moving the recovery code out of the kernel and into a separate `fsck` process.
3. Control returns to `recover_ext3_journal` which then calls `e2fsck_journal_release` which writes the now cleared journal to disk. Unfortunately, the lack of sync barriers allows this write to reach disk before the modified data. As a result, a crash that occurs after this point can obliterate parts of the file system, but the journal will be empty, causing data loss.

```
// e2fsprogs-1.34/e2fsck/jfs_user.h

// Error: empty macro, does not sync data!
#define fsync_no_super(dev) do {} while(0)

// e2fsprogs-1.34/e2fsck/journal.c
static errcode_t recover_ext3_journal(e2fsck_t ctx) {
        journal_t *journal;
        int retval;

        journal_init_revoke_caches();
        retval = e2fsck_get_journal(ctx, &journal);
        /* ... */
        retval = -journal_recover(journal);
        /* ... */

        // Flushes empty journal.
        e2fsck_journal_release(ctx, journal, 1, 0);
        return retval;
}

// e2fsprogs-1.34/e2fsck/recovery.c
int journal_recover(journal_t *journal) {
    // process journal records using cached writes.
    err = do_one_pass(journal, &info, PASS_SCAN);
    if (!err)
      err = do_one_pass(journal, &info, PASS_REVOKE);
    if (!err)
      // writes persistent data recorded in
      // journal using cached write calls.
      err = do_one_pass(journal,&info,PASS_REPLAY);

    /* ... */

    // Write all modified data back before clearing journal.
    fsync_no_super(journal->j_fs_dev);
    return err;
}
```

Figure 4: Journal write ordering bug in ext3 `fsck`.

When this bug was reported, the developers immediately released a patch. ReiserFS and JFS both had similar bugs (both now fixed), but in these systems the code lacked any attempt to order the journal clear with the writes of journal data.

**Buggy transaction abort and `fsck`**. There were five bugs of this type, all in JFS. Their causes were threefold.

First, JFS immediately applies all journaled operations to its in-memory metadata pages. Unfortunately, doing so makes it hard to roll back aborted transactions since their modifications may be interleaved with the writes of many other ongoing or committed transactions. As a result, when JFS aborts a transaction, it relies on custom code to carefully extricate the side-effects of the aborted transactions from non-aborted ones. If the writer of this code forgets to reverse a modification, it can be flushed to disk, interlacing many directories with invalid entries from aborted transactions.

Second, JFS's `fsck` makes no attempts to recover any valid entries in such directories. Instead its recovery policy is that if a directory contains a single invalid entry it will remove all the entries of the directory and attempt to reconnect subdirectories and files into "`lost+found`." This opens a huge vulnerability: any file system mistake that results in persistently writing an invalid entry to disk will cause `fsck` to deliberately destroy the violated directory.

Third, JFS `fsck` has an incorrect optimization that allows the loss of committed subdirectories and files. JFS dynamically allocates and places inodes for better performance, tracking their location using an "inode map." For speed, incremental modifications to this map are written to the on-disk journal rather than flushing the map to disk on every inode allocation or deletion. During reconstruction, the `fsck` code can cause the loss of inodes because while it correctly applies these incremental modifications to its copy of the inode map, it deliberately does not overwrite the out-of-date, on-disk inode map with its (correct) reconstructed copy.

Figure 5 shows this bug, which has been in the JFS code since the initial version of JFS `fsck` over three years ago. The implementors incorrectly believed that if the file system was marked dirty, flushing the inode map was unnecessary because it would be rebuilt later. While the fix is trivial (always flushing the map), this bug was hard to find without a model checker; the JFS developers believe they have been chasing manifestations of it for a while [23]. After we submitted the bug report with all the file system events (operations and sector writes) and choices made by the model checker, a JFS developer was able to create a patch in a couple of days. This was a good example of the fact that model checking improves on testing by being more systematic, repeatable, and better controlled.

**Other data loss bugs**. A JFS journal that spans three or more sectors has the following vulnerability. JFS stores a sequence number in both the first and last sector of its journal but not in the middle sectors. After a crash, JFS `fsck` checks that these sequence numbers match and, if so, replays the journal. Without additional checking, inopportune sector reorderings can obviously lead to a corrupt journal, which will badly mutilate the file system when replayed.

Both JFS and ext3 had a bug where a crashed file system's superblock could be falsely marked as "clean." Thus, their `fsck` program would not repair the system, potentially leading to data loss or a system crash.

```
// jfsutils-1.1.5/libfs/logredo.c
/* [Original, incorrect comment]
 *  don't update the maps if the aggregate/lv is
 *  FM_DIRTY since fsck will rebuild maps anyway */
if (!vopen[k].is_fsdirty) {// check dirtiness
    // update on-disk map
    if ((rc = updateMaps(k)) != 0) {
        fsck_send_msg(lrdo_ERRORCANTUPDMAPS);
        goto error_out;
    }
}
```

Figure 5: Incorrect JFS `fsck` optimization which causes unrecoverable loss of inodes and their associated data.

The last data loss bug happened when JFS incorrectly stored a negative error code as an inode number in a directory entry; this invalid entry would cause any later `fsck` invocation to remove the directory.

## 7.2 Security Holes

While we did not target security, FiSC found five security holes, three of which appear readily exploitable.

The easiest exploit we found was a storage leak in the JFS routine `jfs_link` used to create hard links. It calls the routine `get_UCSname`, which allocates up to 255 bytes of memory. `jfs_link` must (but does not) free this storage before returning. This leak occurs each time `jfs_link` is called, allowing a user to trivially do a denial of service attack by repeatedly creating hard links. Even ignoring malice, leaking storage on each hard link creation is generally bad.

The two other seemingly exploitable errors both occurred in `ext2` and were both caused by lookup routines that did not distinguish between lookups that failed because (1) no entry existed or (2) memory allocation failed. The first bug allows an attacker to create files or directories with the same name as a preexisting file or directory, hijacking all reads and writes intended for the original file. The second allows a user to delete nonempty directories to which they do not have write access.

In the first case, before creating a new directory entry, ext2 will call the routine `ext2_find_entry` to see if the entry already exists. If `ext2_find_entry` returns NULL, the directory entry is created, otherwise it returns an error code. Unfortunately, in low memory conditions `ext2_find_entry` can return NULL even if the directory entry exists. As shown in Figure 6, the routine iterates over all pages in a directory. If page allocation fails (`ext2_get_page` returns NULL) it will skip this directory worth of entries and go to the next. Under low memory, `ext2_get_page` will always fail, no entry will be checked, and `ext2_find_entry` will always return NULL. This allows a user with write ac-

```
// linux-2.4.19/ext2/dir.c
struct ext2_dir_entry_2 * ext2_find_entry (struct inode * dir,
            struct dentry *dentry, struct page ** res_page)
{
    unsigned long start, n;
    unsigned long npages = dir_pages(dir);
    struct page *page = NULL;
    /* ... */
    // Iterate through all pages of the directory
    do {
        page = ext2_get_page(dir, n);
        if (!IS_ERR(page)) {
            // Code to check entry existence
            // Return the corresponding entry once found.
            /* ... */
        }
        // BUG: Error return from ext2_get_page ignored
    } while (...);
    return NULL;
    /* ... */
}
```

Figure 6: Ext2 security hole in `ext2_find_entry`.

cess to the directory to effectively create files and subdirectories with the same name as an existing file, hijacking all reads and writes intended for the original file.

The second error was similar, `ext2_rmdir` calls the routine `ext2_empty_dir` to ensure that the target directory is empty. Unfortunately the return value of `ext2_empty_dir` is the same if either the directory has no entries or if memory allocation fails, allowing an attacker to delete non-empty directories when they should not have permission to do so.

The remaining two errors occurred in ext3 and were identical to the ext2 bugs except that they were caused by disk read errors rather than low-memory conditions.

## 7.3 Other Bugs

**Kernel crashes.** There were 12 bugs which caused the kernel to crash because of a null pointer dereference. Most of these errors were due to improperly handled allocation failures. There was one error in the VFS layer, one error in ReiserFS, and 10 in JFS. The most interesting error was in JFS where `fsck` failed to correctly repair a file system, but marked it as clean. A subsequent traversal of the file system would panic the kernel.

**Incorrect code.** There were two cases where code just did the wrong thing. For example, `sys_create` creates a file on disk, but returns an error if a subsequent allocation fails. The application will think the file has not been created when it has. This error was interesting since it was in very heavily tested code in the VFS layer shared by all file systems.

**Leaks.** In addition to the leak mentioned above, the JFS routine `jfs_unmount` leaks memory on every unmount of a file system.

# 8 Experience

This section describes some of our experiences with FiSC: its use during development, sources of false positives and false negatives, and design lessons learned.

## 8.1 FiSC-Assisted Development

We checked preexisting file systems, and so could not comprehensively study how well model checking helps the development process. However, the responsiveness of the JFS developers allowed us to do a micro-case study of FiSC-assisted software development by following the evolution of a series of mistaken fixes:

1. We found and reported two kernel panics in the JFS transaction abort function `txAbortCommit` when called by the transaction commit function `txCommit` if memory allocation failed.
2. A few days later, the JFS developers sent a patch that removed `txAbortCommit` entirely and made `txCommit` call `txAbort` instead.
3. We applied the patch and replayed the original model checking sequence and verified it fixed the two panics. However, when we ran full model checking, within seconds we got segmentation faults in the VFS code. Examination revealed that the newly created inode was inserted into the VFS directory entry cache before the transaction was committed. A failed commit freed the inode and left a dangling pointer in the VFS directory entry cache. We sent this report back to the JFS developers.
4. As before: a few days later, they replied with a second patch, we applied it, it again fixed the specific error that occurred. We ran FiSC on the patched code and found a new error, where `fsck` would complain that a parent directory contained an invalid entry and would remove the parent directory entirely. This was quite a bit worse than the original error.
5. This bug is still outstanding.

While there are many caveats that one must keep in mind, model checking has some nice properties. First, it makes it trivial to verify that the original error is fixed. Second, it allows more comprehensive testing of patches than appears to be done in commercial software houses. Third, it finds the corner-case implications of seemingly local changes in seconds and demonstrates that they violate important consistency invariants.

## 8.2 False Positives

The false positives we found fell into two groups. Most were bugs in the model checking harness or in our understanding of the underlying file system and not in the checked code itself. The latter would hopefully be a minor problem for file system implementors using our system (though it would be replaced by problems arising from their imperfect understanding of the underlying model checker). We have had to iteratively correct a series of slight misunderstandings about the internals of each of the file systems we have checked.

The other group of false positives stemmed from implementors intentionally ignoring or violating the properties we check. For example, ReiserFS causes a kernel panic when disk read fails in certain circumstances. Fortunately, such false positives are easily handled by disabling the check.

## 8.3 False Negatives

In the absence of proving total correctness, one can always check more things. We are far from verification. We briefly describe what we believe are the largest sources of missed errors.

**Exploring thresholds**. We do a poor job of triggering system behavior that only occurs after crossing a threshold value. The most glaring example: because we only test a small number of files and directories ($\leq 15$) we miss bugs that happen when directories undergo reorganization or change representations only after they contain a "sufficient" number of entries. Real examples include the re-balancing of directory tree structures in JFS or using a hashed directory structure in ext3. With that said, FiSC does check a mixture of large and small files (to get different inode representations) and file names or directories that span sector boundaries (for crash recovery).

**Multi-threading support**. The model checker is single-threaded both above and below the system call interface. Above, because only a single user process does file system operations. Below, because each state transition runs atomically to completion. This means many interfering state modifications never occur in the checked system. In particular, in terms of high-level errors, file system operations never interleave and, consequently, neither do partially completed transactions (either in memory or on disk). We expect both to be a fruitful source of bugs.

**White-box model checking**. FiSC can only flag errors that it sees. Because it does not instrument code it can miss low-level errors, such as memory corruption, use of freed memory, or a race condition unless they cause a crash or invariant violation. Fortunately, because we model-check implementation code we can simultaneously run dynamic tools on it.

**Unchecked guarantees**. File systems provide guarantees that are not handled by our current framework. These include versioning, undelete operations, disk quotas, access control list support, and journaling of data or, in fact, any reasonable guarantees of data block contents across crashes. The latter is the one we would like to fix the most. Unfortunately, because of the lack of agreed-upon guarantees for non-`sync`'d data across crashes we currently only check metadata consistency across crashes — data blocks that do not precede a "sync" point can be corrupted and lost without complaint.

File systems are directed acyclic graphs, and often trees. Presumably events (file system operations, failures, bad blocks) should have topological independence — events on one subgraph should not affect any other disjoint subgraph. Events should also have temporal independence in that creating new files and directories should not harm old files and directories.

One way to broaden the invariants we check would be to infer FS-specific knowledge using the techniques in [29].

**Missed states**. While our state hashing (§5.1) can potentially discard too much detail, we do not currently discard enough of the right details, possibly missing real errors. Using FS-specific knowledge opens up a host of additional state optimizations. One profitable example would be if we knew which interleavings of buffer cache blocks and `fsck` written blocks are independent (e.g., those for different files), which would dramatically reduce the number of permutations needed for checking the effects of a crash.

We have not aggressively verified statement coverage, so all file systems almost certainly contain many unexercised statements.

## 8.4 Design Lessons

One hard lesson we learned was a sort of "Heisenberg" principle of checking: make sure the inspection done by your checking code does not perturb the state of the checked system. Violating this principle leads to mysterious bugs. A brief history of the code for traversing a mounted file system and building a model drives this point home.

Initially, we extracted the VolatileFS by using a single block device that the test driver first mutated and then traversed to create a model of the volatile file system after the mutation. This design deadlocked when a file system operation did a multi-sector write and the traversal code tried to read the file system after only one of the sectors was written. The file system code responsible for the write holds a lock on the file being written, a lock that the traversal code wants to acquire but cannot. We removed this specific deadlock by copying the disk after a test driver operation and then traversing this copy, essentially creating two file systems. This hack worked until we started exploring larger file system topologies, at which point we would deadlock again because the creation of the second file system copy would use all available kernel memory, preventing the traversal thread from being able to successfully complete. Our final hack to solve this problem was to create a reserve memory pool for the traversal thread.

In retrospect, the right solution is to run two kernels side by side: one dedicated to mutating the disk, the other to inspecting the mutated disk. Such isolation would straightforwardly remove all perturbations to the checked system.

A similar lesson is that the system being checked should be instrumented instead of modified unless absolutely necessary. Code always contains hidden assumptions, easily violated by changing code. For example, the kernel we used had had its kernel memory allocators re-implemented in previous work [25] as part of doing leak checking. While this replacement worked fine in the original context of checking TCP, it caused the checked file systems to crash. It turned out they were deliberately mangling the address of the returned memory in ways that intimately depended on how the original allocator (`page_alloc`) worked. We promptly restored the original kernel allocators.

## 9 Related Work

In this section, we compare our approach to file system testing techniques, software model checking efforts and other generic bug finding approaches.

**File system testing tools.** There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment. While these frameworks are less comprehensive than model checking they require much less work than that required to jam an entire OS into a model checker. We view testing as complementary to model checking — there is no reason not to test a file system and then apply model checking (or vice versa). It is almost always the case that two different but effective tools will find different errors, irrespective of their theoretical strengths and weaknesses.

**Software Model Checking.** Model checkers have been previously used to find errors in both the design and the implementation of software systems [1, 3, 6, 18–20].

We compare our work with two model checkers that are the most similar to our approach, both of which execute the system implementation directly without resorting to an intermediate description.

Verisoft [18] is a software model checker that systematically explores the interleavings of a concurrent C program. Unlike the CMC model checker we use, Verisoft does not store states at checkpoints and thereby can potentially explore a state more than once. Verisoft relies heavily on partial order reduction techniques that identify (control and data) independent transitions to reduce the interleavings explored. Determining such independent transitions is extremely difficult in systems with tightly coupled threads sharing large amount of global data. As a result, Verisoft would not perform well for these systems, including the Linux file systems checked in this paper.

Java PathFinder [3] is very similar to CMC and systematically checks concurrent Java programs by checkpointing states. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. The techniques described in this paper are applicable to Java Pathfinder as well.

**Generic bug finding.** There has been much recent work on bug finding, including both better type systems [9, 14, 16] and static analysis tools [1, 4, 7, 8, 11, 15]. Roughly speaking [12], because static analysis can examine all paths and only needs to compile code in order to check it, it is relatively better at finding errors in surface properties visible in the source ("`lock` is paired with `unlock`"). In contrast, model checking requires running code, which makes it much more strenuous to apply (days or weeks instead of hours) and only lets it check executed paths. However, because it executes code it can more effectively check the properties implied by code. (E.g., that the log contains valid records, that `fsck` will not delete directories it should not.) Based on our experiences using static analysis, the most serious errors in this paper would be difficult to get with that approach. But, as with testing, we view static analysis as complementary to model checking — it is lightweight enough that there is no reason not to apply it and then use model checking.

## 10 Conclusion

This paper has shown how model checking can find interesting errors in real file systems. We found 32 serious errors, 10 of which resulted in the loss of crucial metadata, including the file system root directory "`/`". The majority of these bugs have resulted in immediate patches.

Given how heavily-tested the file systems we model-checked were and the severity of the errors found, it appears that model checking works well in the context of file systems. This was a relief — we have applied full system model-checking in other contexts less successfully[12]. The underlying reason for its effectiveness in this context seems to be because file systems must do so many complex things right. The single worst source of complexity is that they must be in a recoverable state in the face of crashes (e.g., power loss) at every single program point. We hope that model checking will show similar effectiveness in other domains that must reason about a vast array of failure cases, such as database recovery protocols and optimized consensus algorithms.

## 11 Acknowledgments

We thank Dave Kleikamp for answering our JFS related questions, diagnosing bugs and submitting patches, Andreas Dilger, Theodore Ts'o, Al Viro, Christopher Li, Andrew Morton, Stephen C. Tweedie for their help with ext2 and ext3, Oleg Drokin and Vitaly Fertman for ReiserFS. We especially thank Ted Kremenek for last-minute comments and edits. We are also grateful to Andrew Myers (our shepherd), Ken Ashcraft, Brian Gaeke, Lea Kissner, Ilya Shpitser, Xiaowei Yang, Monica Lam and the anonymous reviewers for their careful reading and valuable feedback.

## References

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[2] H.-J. Boehm. Simple garbage-collector-safety. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[7] SWAT: the Coverity software analysis toolset. `http://coverity.com`.

[8] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[9] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[10] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525. IEEE Computer Society, 1992.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, Sept. 2000.

[12] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Invited paper: Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04)*, pages 191–210, Jan. 2004.

[13] The ext2/ext3 File system. `http://e2fsprogs.sf.net`.

[14] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[15] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[16] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[17] G. R. Ganger and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. Technical report, University of Michigan, 1995.

[18] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[19] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[20] G. J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., 2001.

[21] The IBM Journaling File System for Linux. `http://www-124.ibm.com/jfs`.

[22] M. K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[23] D. Kleikamp. Private communication.

[24] Linux Test Project. `http://ltp.sf.net`.

[25] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[26] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[27] ReiserFS. `http://www.namesys.com`.

[28] Sandberg, Goldberg, Kleiman, Walsh, and Lyon. Design and implementation of the Sun network file system, 1985.

[29] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Second USENIX Conference on File and Storage Technologies*, 2003.

[30] stress. `http://weather.ou.edu/~apw/projects/stress`.

[31] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.