Model-Checking Support for File System Development

Wei Su, Yifei Liu, Gomathi Ganesan
Stony Brook University

Gerard Holzmann
Nimble Research

Scott Smolka, Erez Zadok
Stony Brook University

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:
Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. 2021. Model-Checking Support for File System Development. In *13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21), July 27–28, 2021, Virtual, USA.* ACM, New York, NY, USA, 8 pages. *https://doi.org/10.1145/3465332.3470878*

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.

HotStorage '21, July 27–28, 2021, Virtual, USA

ACM ISBN 978-1-4503-8550-3/21/07...$15.00
https://doi.org/10.1145/3465332.3470878

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.

CMC [29–31] inserts file system code directly into the model checker, for substantial speed advantages, but requires extensive changes to the *very code being verified*, making the results less trustworthy [47]. Fuzzing can find real-world bugs [12, 22, 38, 45, 46], but either is limited to specific types of bugs (*e.g.*, memory safety for Janus [46]), or needs human effort to create checkers [22]. Likewise, symbolic-execution tools [4, 5] cannot guarantee thorough coverage because they focus on particular issues (*e.g.*, corrupt input [5]); they also require a behavioral model of each file system call [4].

2 MCFS DESIGN

Model checking is an automatic method for verifying finite-state concurrent systems. It performs an exhaustive search of the state space to verify whether a system's model conforms to its specification. We exploit this efficient search technology to explore file system states exhaustively. The MCFS model-checking framework is designed to detect *discrepant behaviors* in file systems; i.e., situations where two file systems behave differently given the same inputs.

MCFS has five key design goals: **(1) Thorough coverage:** It should thoroughly explore the state spaces of the file systems we check, so that it can uncover as many corner cases as possible. This requires MCFS to check as many permutations of file system operations as possible, exploring all possible changes that the given set of operations can make. **(2) Eliminate need for an abstract model:** Traditional model checking requires one to define a model for the system under investigation. MCFS can verify file systems without a model because it directly executes system code to perform state-space exploration.

(3) Absence of observer effects: To ensure accuracy, MCFS should avoid changing the behavior of the investigated file system, ideally treating it as a black box. If we have to modify it, we must be careful to minimize MCFS's footprint. **(4) Universality:** The model-checking framework should support a wide range of file systems (*e.g.*, in-kernel, user-level, networked, distributed). **(5) High performance:** Since a file system's state space is the product of many system calls and their parameters, the number of states can be exponential. The model checker must be able to enumerate new states as fast as possible, exploring large fractions of the state space within a reasonable amount of time.

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 `atime` timestamps and the physical locations of data blocks. Hashing the entire 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 CHALLENGES

We now describe challenges we encountered while developing MCFS, and our attempts to work around them.

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

```
Input  : Path to the file system mount point path
Output : 128-bit MD5 Hash
files ← list ([])
md5ctx ← md5_init ()
// Recursively walk the mount-point directory
foreach file in recursively_traverse_dir (path) do
    files.append (file)
sort (files) // Sort files by pathnames
foreach file in files do
    fd ← open (file.path)
    content ← read (fd) // Read all file content
    md5_update (md5ctx, content)
    close (fd)
    attrs ← stat (file)
    // Get important metadata
    attrs' ← important_attributes (attrs)
    md5_update (attrs')
    md5_update (file.path)
return get_md5_hash (md5ctx)
```

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 as S_L; then on each file system with free space S_n, it creates a dummy file and writes $S_n - S_L$ 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 `mmap`s 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* `create`s and then `close`s a file; *write_file* `open`s, `write`s some data to, and `close`s 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

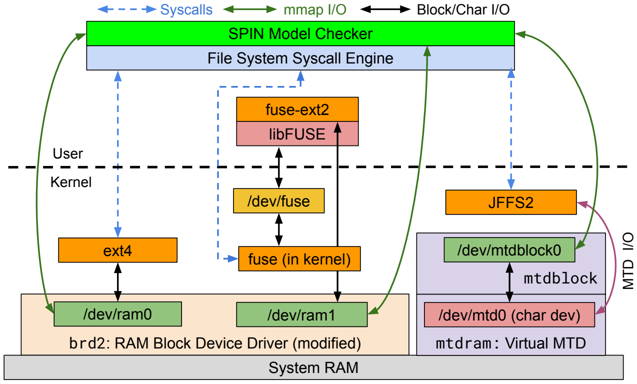

Figure 1: Model checking workflow for different file system types: from left to right, block-based, FUSE, and character-device-based.

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 incoherency. 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 `ioctl`s: `ioctl_CHECKPOINT` and `ioctl_RESTORE`.

When MCFS calls `ioctl_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, `ioctl_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.

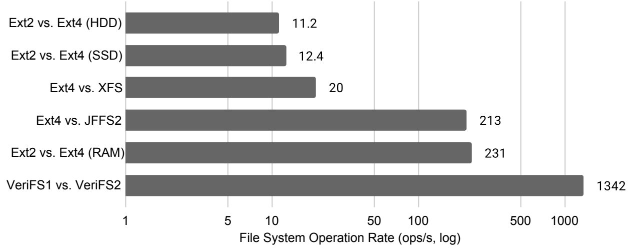

Figure 2: Speed comparison for different experiments. Unless specified in parentheses, all experiments were run on RAM disks or entirely in memory.

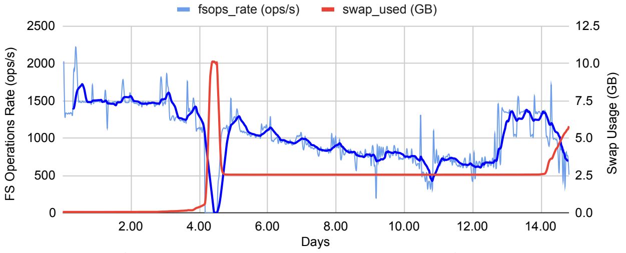

Figure 3: File system operation rate and swap usage in a two-week MCFS experiment on VeriFS1.

We ran MCFS with Ext4 and VeriFS1 for over 5 days; MCFS executed over 159 million syscalls 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, 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 content. The bug arose when `truncate` failed to clear newly allocated space when expanding a file. The second bug was detected after about 12K operations; MCFS created a test directory in Ext4 but VeriFS failed, claiming that the directory existed—but in fact it did not. This was due to cache incoherency between the kernel and VeriFS's in-memory state. When VeriFS rolled back to an earlier state, the kernel's inode and dentry caches did not keep up. The fix was to call FUSE's cache-invalidation APIs (`fuse_lowlevel_notify_inval_entry` and `fuse_lowlevel_notify_inval_inode`).

We then developed VeriFS2 (see Section 5) and used MCFS to assist development by model-checking it vs. VeriFS1. MCFS found two more bugs during this phase. The first occurred after over 900K operations, when the test files in VeriFS1 and VeriFS2 had different data. VeriFS2 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 `ioctls`. These functions let MCFS track VeriFS's full state while avoiding cache incoherency. 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

[1] Alper Akcan. Fuse-ext2 GitHub repository, 2021. *https://github.com/alperakcan/fuse-ext2.*

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, April 2016.

[3] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, April 2016.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[5] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, Alexandria, VA, November 2006.

[6] Tej Chajed. Verifying an I/O-concurrent file system. Master's thesis, Massachusetts Institute of Technology, 2017.

[7] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.

[9] Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, Eddie Kohler, and Nickolai Zeldovich. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[10] CRIU Community. Checkpoint/restore in userspace (CRIU), 2021. *https://criu.org/*.

[11] Ext4. *http://ext4.wiki.kernel.org/*.

[12] Google. syzkaller: Linux syscall fuzzer, 2021. *https://github.com/google/syzkaller*.

[13] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015.

[14] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.

[15] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[16] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 131–147, Berlin, Heidelberg, 2000. Springer-Verlag.

[17] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6. IEEE, 2008.

[18] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2010.

[19] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Formal Description Techniques VII*, pages 197–211. Springer, 1995.

[20] Atalay Mert Ileri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–338, Carlsbad, CA, October 2018.

[21] Kernel.org Bugzilla. Btrfs bug entries, 2021. *https://bugzilla.kernel.org/buglist.cgi?component=btrfs*.

[22] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 147–161, Toronto, Ontario, Canada, October 2019.

[23] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 192–203, Gothenburg, Sweden, July 2001.

[24] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2013. USENIX Association.

[25] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 218–233, Shanghai, China, October 2017.

[26] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 361–377, Monterey, CA, October 2015.

[27] Subrata Modak. Linux test project (LTP), 2009. *http://ltp.sourceforge.net/*.

[28] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50, Carlsbad, CA, October 2018.

[29] Madanlal Musuvathi, Andy Chou, David L. Dill, and Dawson R. Engler. Model checking system software with CMC. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 219–222, Saint-Emilion, France, July 2002.

[30] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 155–168, San Francisco, CA, March 2004.

[31] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, December 2002.

[32] NFS-Ganesha, 2016. *http://nfs-ganesha.github.io/*.

[33] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, October 2014.

[34] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220,

Brighton, UK, October 2005. ACM Press.

[35] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 38–53, Monterey, CA, October 2015.

[36] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[37] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009.

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security)*, pages 167–182, Vancouver, BC, Canada, August 2017.

[39] SGI XFS. xfstests, 2016. *http://xfs.org/index.php/Getting_the_latest_source_code*.

[40] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[41] Spin Project. Static analysis tools for C code. *www.spinroot.com/static/*.

[42] Miklos Szeredi. Filesystem in Userspace. *http://fuse.sourceforge.net*, February 2005.

[43] David Woodhouse, Joern Engel, Jarkko Lavinen, and Artem Bityutskiy. General MTD documentation, 2009.

[44] David Woodhouse, Joern Engel, Jarkko Lavinen, and Artem Bityutskiy. JFFS2, 2009.

[45] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313–2328, Dallas, TX, October 2017.

[46] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 818–834, San Francisco, CA, May 2019.

[47] Junfeng Yang, Can Sar, and Dawson Engler. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006.

[48] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, December 2004.

[49] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(2):161–196, 2006.

[50] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 259–274, Toronto, Ontario, Canada, October 2019.