# Model-Checking Support for File System Development

## 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)

Wei Su[1], Yifei Liu[1], Gomathi Ganesan[1], Gerard Holzmann[2], Scott Smolka[1], Erez Zadok[1] , and Geoff Kuenning[3]

[1] Stony Brook University; [2] Nimble Research; [3] Harvey Mudd College

# Background and Motivation

- File system development
  - Time-consuming: many years of effort
  - Complex: standards compliance, concurrency, etc.
  - Error-prone: bugs and defects

- File system bugs
  - Constantly emerging
  - Serious consequences
    - Data loss
    - System crashes
    - Data corruption

| File System | Age (years) | # of bugs in 2020 |
|---|---|---|
| Btrfs | 14 | 110 |
| Ext4 | 15 | 17 |
| XFS | 18 | 30 |

# Existing Work on File System Bugs

| Regression Test Suites | Fuzzing | Machine-Verifiable File Systems | Model Checking |
|---|---|---|---|
| Linux Test Project xfstests | kAFL (Security '17)<br>Janus (S&P '19)<br>Hydra (SOSP '19)* | FSCQ (SOSP '15)<br>Yggdrasil (OSDI '16)<br>DFSCQ (SOSP '17)<br>SFSCQ (OSDI '18)<br>AtomFS (SOSP '19) | CMC (OSDI '02)<br>FiSC (OSDI '04)<br>eXplode (OSDI '06)<br>JUXTA (SOSP '15)<br>Ferrite (ASPLOS '16)<br>$B^3$ (OSDI '18)<br>**MCFS (Our work)** |
| **Cannot sufficiently cover corner cases and new features** | **Only detect certain types of bugs or need developers to write their own checkers** | **Cannot apply to existing file systems directly** | **Only find certain types of bugs or need effort to build an abstract file system model** |

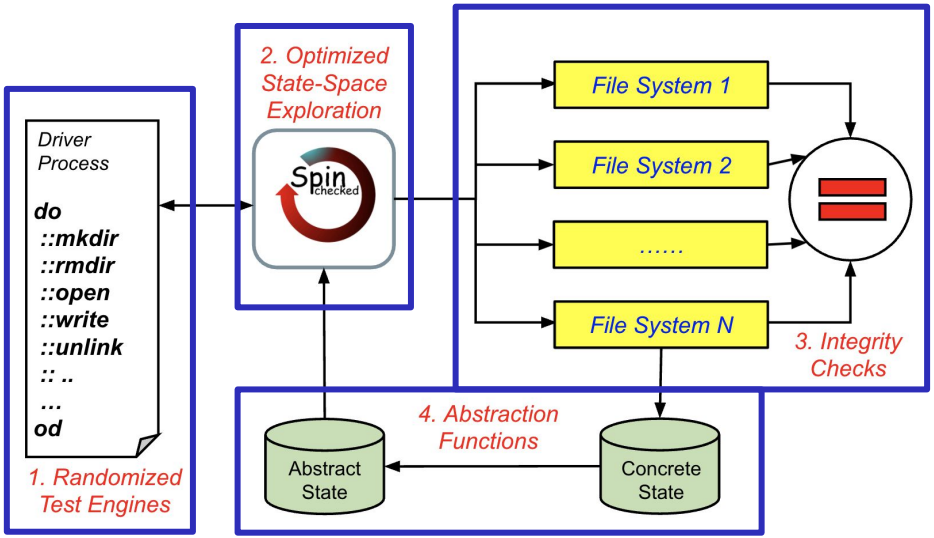
*Kim et al. "Finding semantic bugs in file systems with an extensible fuzzing framework." Proceedings of 27th ACM Symposium on Operating Systems Principles. 2019.

# MCFS Design Goals

1. Thorough coverage
   - Explore (bounded) state spaces of file systems thoroughly
2. Eliminate the need for an abstract model
   - Check file systems without building a model
3. Absence of observer effects
   - Avoid changing the behavior of the investigated file system
4. Universality
   - Support a wide range of file systems
5. High performance
   - Perform state exploration efficiently

# MCFS Architecture

1. Randomized test engines
   - Issue syscall sequences to each file system
2. Optimized state-space exploration
   - Rely on Spin to perform state-space exploration
3. Integrity checks
   - Verify all file systems have identical states
4. Abstraction functions
   - Convert concrete states into abstract states



# Challenges

- State Explosion
- False Positives
- Cache Incoherency

# Challenge 1: State Explosion

## Challenges

- Spin compares states byte-to-byte
- Any small change in the file system image will be considered a new state
  - ➜ E.g., Updates to timestamps in ext4's super block
- State exploration will never complete

## Solutions

- Define "abstraction functions" to calculate abstract states
- Only consider directory structure, file content, and important metadata
- Ignore noisy attributes, such as timestamps and locations of data blocks

# Challenge 2: False Positives

## Challenges

- Size reporting of directories
- Ordering of directory entries
- Special files and folders
- Different file data capacity exposed

## Solutions

- Ignore sizes of non-regular files when calculating abstract states
- Sort all paths after recursively walking the file systems
- Use an “exclusion list”: ignore these special files and folders when walking the file systems
- Write dummy data to equalize available space of all file systems

# Challenge 3: Cache Incoherency

## Challenges

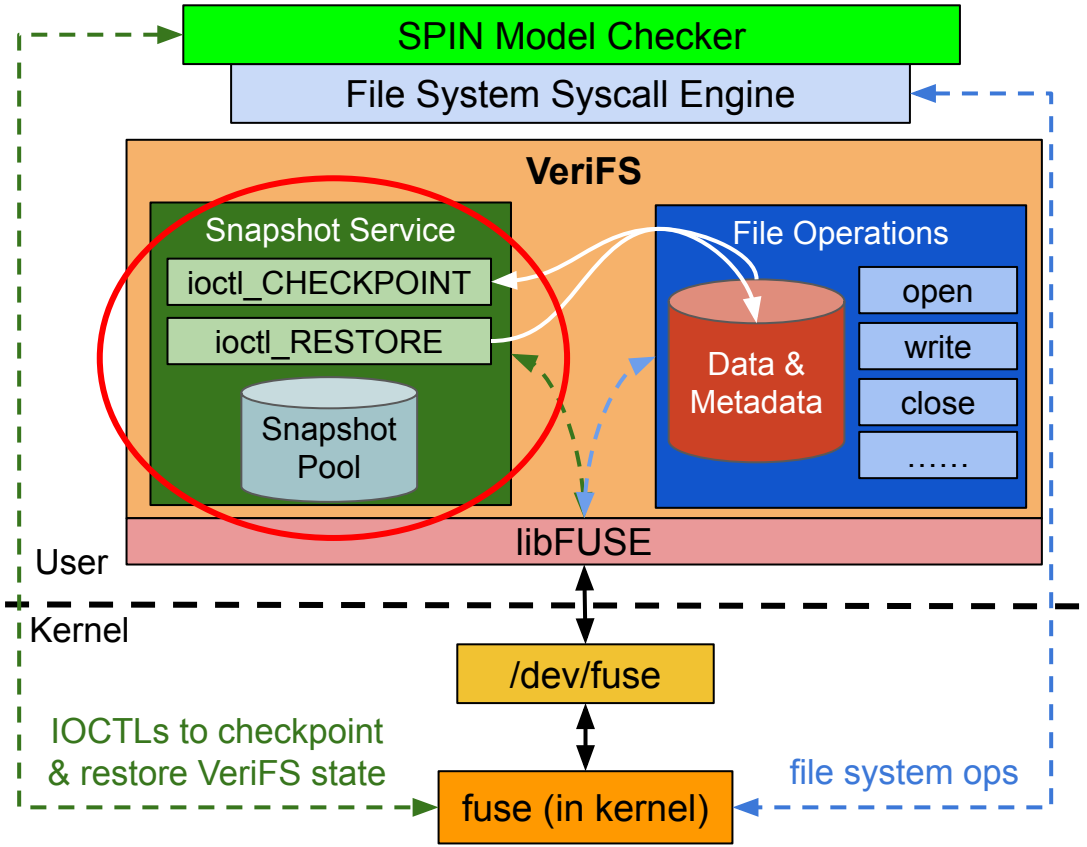
- Spin needs to checkpoint and restore the states of file systems when exploring in BFS or DFS order
- However, Spin cannot track the in-memory states of the file systems
  - ➜ Kernel FS: Spin is a user-space program
  - ➜ FUSE FS: Independent processes, independent address spaces
- Leads to: incomplete state restoration when backtracking → corrupted file system state

## Solutions

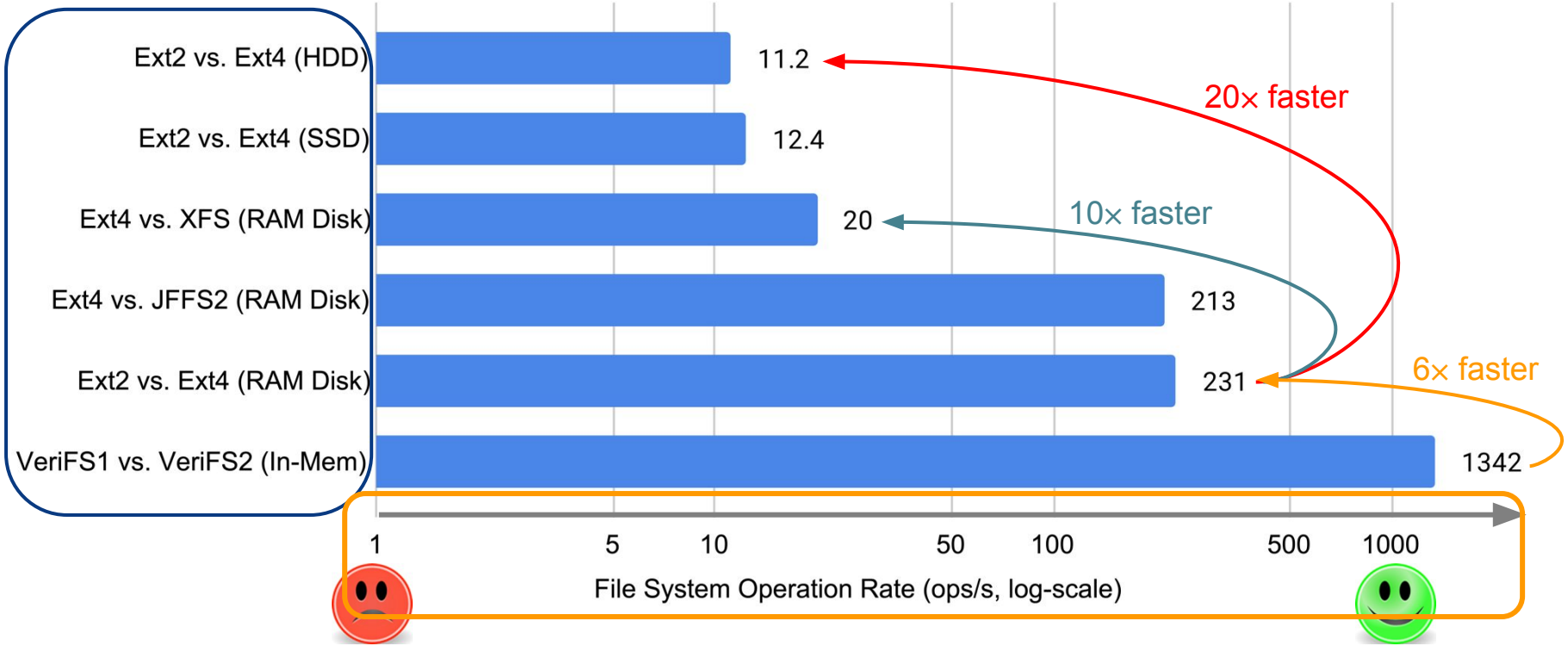
- Remount / unmount the file systems before / after running each syscall
  - ➜ Slow
  - ➜ Hide bugs related to in-memory states
  - ➜ Compatible with all on-disk file systems
- Checkpoint & restore API
  - ➜ Efficient and preserves in-memory states
  - ➜ Needs to implement them

# VeriFS and Checkpoint/Restore API

- `ioctl_CHECKPOINT`:
  - Save current concrete file system state as a snapshot
- `ioctl_RESTORE`:
  - Restore file system state from a snapshot
- Snapshot Pool
  - Key-value store of snapshots
  - 64-bit key and corresponding file system snapshot
- Two versions of VeriFS
  - VeriFS2 uses more sophisticated data structures



# Evaluation: State Exploration Speed



# MCFS's Ability to Detect Bugs

- VeriFS1 vs. Ext4
  - VeriFS1 had different file content from Ext4
    - VeriFS1's `truncate()` failed to zero the new space if it expanded the file
- VeriFS1 vs. VeriFS2
  - VeriFS2 had different file content from VeriFS1
    - Failed to zero the file buffer if `write` created a hole

# Conclusions and Future Work

- MCFS: model-checking framework for file system development
  - Thoroughly explore file system states
  - Developed VeriFS1 and VeriFS2 and their checkpoint/restore ioctls
  - Can find real bugs and assist file system development

- Future work
  - Add the checkpoint/restore API to Linux VFS, thereby eliminating need for unmount/remount workaround
  - Run more than two file systems at a time; apply n-version programming
  - Use Spin's swarm verification to explore larger state spaces in parallel

# Model-Checking Support for File System Development

Wei Su[1], Yifei Liu[1], Gomathi Ganesan[1], Gerard Holzmann[2], Scott Smolka[1], Erez Zadok[1] , and Geoff Kuenning[3]

[1]Stony Brook University; [2]Nimble Research; [3]Harvey Mudd College

# Thank You

# Q&A

# Challenge 3: Cache Incoherency

- Spin applies depth-first search to explore state space
  - When Spin reaches leave nodes, it needs to backtrack to the parent. Backtracking requires restoring the file system states to an earlier version.
  - By default, Spin uses memcpy() to checkpoint and restore the states it tracks.
- File system states
  - Persistent states (the block devices)
    - We can have Spin track them by mmap() the devices
  - In-memory states:
    - Kernel FS: Spin, as a user process, cannot access the kernel memory space
    - FUSE FS: Spin and the file systems reside in independent address spaces
- Cache Incoherency
  - Reason: Only persistent states are restored, in-memory states are left unchanged
  - Symptom: Directory entries with corrupted or zeroed inodes

# Cache Incoherency Work-around Attempts

- Forcibly flush / load in-memory states of file systems
  - Unmounting flushes all in-memory states (caches) into the underlying storage
  - Remounting loads what's in the disks back into memory
  - We unmounted and remounted the file system between executing each syscall
  - Ensured that caches are coherent
- Deficiencies of unmount & remount
  - Considerably slowed state space exploration
  - Prevented MCFS from identifying file-system bugs caused by incorrect in-memory states

# Tracking Full File System States

- Process snapshotting
  - User-space file systems (FUSE) run as independent processes
  - CRIU: refused to checkpoint processes that opened any character or block device
  - Only applicable to user-space file systems, whereas most mature file systems are in kernel
- Virtual-machine (VM) snapshotting
  - Tracks full file system states by snapshotting and resuming the whole VM
  - But is slow and heavyweight (even for LightVM)
- VeriFS with `ioctl`s
  - A file system checkpoints and restores its own full states
  - VeriFS provides checkpoint and restore APIs via `ioctl`s

# Evaluation: Experiment Setup

- Experiment Machine
  - 16 Cores, 64GB RAM
  - 128GB of swap space on a SSD
- File Systems
  - Ext2 (256KB RAM block device)
  - Ext4 (256KB RAM block device / HDD / SSD)
  - JFFS2 (256KB RAM-based MTD device)
  - XFS (16MB RAM block device)
  - VeriFS 1 & 2 (In-memory file systems)

# Evaluation: Performance