CrashMonkey and ACE: Systematically Testing File-System Crash Consistency

JAYASHREE MOHAN, ASHLIE MARTINEZ, SOUJANYA PONNAPALLI, and PANDIAN RAJU, University of Texas at Austin
VIJAY CHIDAMBARAM, University of Texas at Austin and VMWare Research

We present CrashMonkey and Ace, a set of tools to systematically find crash-consistency bugs in Linux file systems. CrashMonkey is a record-and-replay framework which tests a given workload on the target file system by *simulating* power-loss crashes while the workload is being executed, and checking if the file system recovers to a correct state after each crash. Ace automatically generates all the workloads to be run on the target file system. We build CrashMonkey and Ace based on a new approach to test file-system crash consistency: *bounded black-box crash testing* (B^3). B^3 tests the file system in a black-box manner using workloads of file-system operations. Since the space of possible workloads is infinite, B^3 bounds this space based on parameters such as the number of file-system operations or which operations to include, and exhaustively generates workloads within this bounded space. B^3 builds upon insights derived from our study of crash-consistency bugs reported in Linux file systems in the last 5 years. We observed that most reported bugs can be reproduced using small workloads of three or fewer file-system operations on a newly created file system, and that all reported bugs result from crashes after `fsync()`-related system calls. CrashMonkey and Ace are able to find 24 out of the 26 crash-consistency bugs reported in the last 5 years. Our tools also revealed 10 *new* crash-consistency bugs in widely used, mature Linux file systems, 7 of which existed in the kernel since 2014. Additionally, our tools found a crash-consistency bug in a verified file system, FSCQ. The new bugs result in severe consequences like broken rename atomicity, loss of persisted files and directories, and data loss.

CCS Concepts: • **General and reference → Reliability**; • **Software and its engineering → File systems management**; • **Computer systems organization → Reliability**;

Additional Key Words and Phrases: Crash consistency, testing, file systems, bugs

This material is based upon work supported by the NSF under CNS-1751277 and generous donations from VMware, Google, and Facebook.

This article is an extended version of the OSDI'18 paper by J. Mohan et al. [73]. The additional material here includes a more detailed discussion of the file system IO path, caching, and re-ordering of IO requests at the volatile storage device cache, a detailed discussion of a new `subset-replay` mode supported by CrashMonkey, use of LSTM to explore the large space of crash states, and a discussion on the lessons learned from our interaction with file-system developers. The Appendix also contains an elaborate list of all the reproduced bugs and the new bugs found by CrashMonkey and Ace.

The authors Jayashree Mohan and Ashlie Martinez contributed equally.

Authors' addresses: J. Mohan, A. Martinez, S. Ponnapalli, and P. Raju, University of Texas at Austin, 110 Inner Campus Drive, Austin, TX 78705; emails: jaya@cs.utexas.edu, ashmrtn@utexas.edu, {soujanyap95, pandian4mail}@gmail.com; V. Chidambaram, University of Texas at Austin and VMWare Research, 110 Inner Campus Drive, Austin, TX 78705; email: vijay@cs.utexas.edu.

1553-3077/2019/04-ART14 $15.00
https://doi.org/10.1145/3320275

ACM Reference format:
Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2019. CrashMonkey and ACE: Systematically Testing File-System Crash Consistency. *ACM Trans. Storage* 15, 2, Article 14 (April 2019), 34 pages.
https://doi.org/10.1145/3320275

1 INTRODUCTION

A file system is *crash consistent* if it always recovers to a correct state after a crash due to a power loss or a kernel panic. The file-system state is correct if the file system's internal data structures are consistent, and files that were persisted before the crash are not lost or corrupted. When developers added delayed allocation to the ext4 file system [65] in 2009, they introduced a crash-consistency bug that led to widespread data loss [36]. Given the potential consequences of crash-consistency bugs and the fact that even professionally managed datacenters occasionally suffer from power losses [67–70, 90, 92], it is important to ensure that file systems are crash consistent.

Unfortunately, there is little to no crash-consistency testing today for widely used Linux file systems such as ext4, xfs [84], btrfs [80], and F2FS [37]. The current practice in the Linux file-system community is to not do any *proactive* crash-consistency testing. If a user reports a crash-consistency bug, the file-system developers will then *reactively* write a test to capture that bug. Linux file-system developers use `xfstests` [23], an ad-hoc collection of correctness tests, to perform regression testing. `xfstests` contains a total of 482 correctness tests that are applicable to all POSIX file systems. Of these 482 tests, only 26 (5%) are crash-consistency tests. Thus, file-system developers have no easy way of systematically testing the crash consistency of their file systems.

This article introduces a new approach to testing file-system crash consistency: *bounded black-box crash testing* (B^3). B^3 is a black-box testing approach: no file-system code is modified. B^3 works by exhaustively generating workloads within a bounded space, *simulating* a crash after persistence operations like `fsync()` in the workload, and finally testing whether the file system recovers correctly from the crash. We implement the B^3 approach by building two tools, CrashMonkey and Ace. Our tools are able to find 24 out of the 26 crash-consistency bugs reported in the last 5 years, across seven kernel versions and three file systems. Furthermore, the systematic nature of B^3 allows our tools to find *new* bugs: CrashMonkey and Ace find 10 bugs in widely used Linux file systems which lead to severe consequences such as `rename()` not being atomic and files disappearing after `fsync()` and a data loss bug in the FSCQ verified file system [13]. We have reported all new bugs; developers have submitted patches for five, and are working to fix the rest. We formulated B^3 based on our study of all 26 crash-consistency bugs in ext4, xfs, btrfs, and F2FS reported in the last 5 years (Section 3). Our study provided key insights that made B^3 feasible: most reported bugs involved a small number of file-system operations on a new file system, with a crash right after a *persistence point* (a call to `fsync()`, `fdatasync()`, or `sync` that flushes data to persistent storage). Most bugs could be found or reproduced simply by systematic testing on a small space of workloads, with crashes only after persistence points. Note that without these insights which bound the workload space, B^3 is infeasible: there are infinite workloads that can be run on infinite file-system images.

Choosing to crash the system only after persistence points is one of the key decisions that makes B^3 tractable. B^3 does not focus on bugs that arise due to crashes in the *middle* of a file-system operation because file-system guarantees are undefined in such scenarios. Moreover, B^3 cannot reliably assume that the on-storage file-system state has been modified if there is no persistence point. Crashing only after persistence points bounds the work to be done to test crash consistency, and also provides clear correctness criteria: files and directories which were successfully persisted

before the crash must survive the crash and not be corrupted. Though we explore how to simulate crashes in the middle of file-system operations (Section 7), it did not result in finding new bugs. We believe that there are several open challenges that need to be addressed before we can efficiently find bugs due to crashes in the middle of file-system operations.

B^3 *bounds* the space of workloads in several other ways. First, B^3 restricts the number of file-system operations in the workload, and simulates crashes only after persistence points. Second, B^3 restricts the files and directories that function as arguments to the file-system operations in the workload. Finally, B^3 restricts the initial state of the system to be a small, new file system. Together, these bounds greatly reduce the space of possible workloads, allowing CrashMonkey and Ace to exhaustively generate and test workloads.

An approach like B^3 is only feasible if we can *automatically* and *efficiently* check crash consistency for arbitrary workloads. We built CrashMonkey, a framework that simulates crashes during workload execution and tests for consistency on the recovered file-system image. CrashMonkey first profiles a given workload, capturing all the IO resulting from the workload. It then replays IO requests until a persistence point to create a new file-system image we term a *crash state*. At each persistence point, CrashMonkey also captures a snapshot of files and directories which have been explicitly persisted (and should therefore survive a crash). CrashMonkey then mounts the file system in each crash state, allows the file system to recover, and uses its own fine-grained checks to validate if persisted data and metadata are available and correct. Thus, CrashMonkey is able to check crash consistency for arbitrary workloads automatically, without any manual effort from the user. This property is key to realizing the B^3 approach.

We built the Automatic Crash Explorer (Ace) to exhaustively generate workloads given user constraints and file-system semantics. Ace first generates a sequence of file-system operations; e.g., a `link()` followed by a `rename()`. Next, Ace fills in the arguments of each file-system operation. It then exhaustively generates workloads where each file-system operation can optionally be followed by an `fsync()`, `fdatasync()`, or a global `sync` command. Finally, Ace adds operations to satisfy any dependencies (e.g., a file must exist before being renamed). Thus, given a set of constraints, Ace generates an exhaustive set of workloads, each of which is tested with CrashMonkey on the target file system.

B^3 offers a new point in the spectrum of techniques addressing file-system crash consistency, alongside verified file systems [12, 13, 82] and model checking [94, 95]. Unlike these approaches, B^3 targets widely deployed file systems written in low-level languages, and does not require annotating or modifying file-system code.

However, B^3 is not without limitations as it is not guaranteed to find all crash-consistency bugs. Currently, Ace's bounds do not expose bugs that require a large number of operations or exhaustion of file-system resources. While CrashMonkey can test such a workload, Ace will not be able to automatically generate the workload. Despite these limitations, we are hopeful that the black-box nature and ease-of-use of our tools will encourage their adoption in the file-system community, unlike model checking and verified file systems. We are encouraged that researchers at Hanyang University are using our tools to test the crash consistency of their research file system, BarrierFS [93].

This article makes the following contributions:

- A detailed analysis of crash-consistency bugs reported across three widely used file systems and seven kernel versions in the last 5 years (Section 3).
- The bounded black-box crash testing approach (Section 4).
- The design and implementation of CrashMonkey and Ace.[1] (Section 5)

[1] https://github.com/utsaslab/crashmonkey.

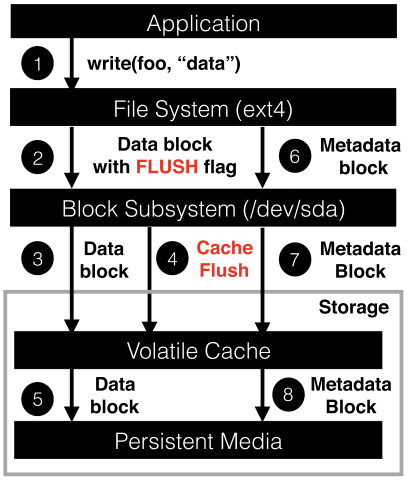

Fig. 1. **Life of a file-system write.** The figure shows an application write being processed by the storage stack. For correctness, the data for the file needs to be persisted before the metadata. The file system sets the FLUSH flag when submitting the request to the block subsystem; the block subsystem flushes the data block before persisting the metadata block.

—Experimental results demonstrating that our tools are able to efficiently find existing and new bugs across widely used Linux file systems and verified file systems. (Section 6)
—Generalizing CrashMonkey to find bugs due to crashes in the middle of file-system operations and demonstrating the use of LSTM to tackle the issue of a large state space. (Section 7)

2 BACKGROUND

We first provide some background on how file systems perform reads and writes, and how those operations are processed through the storage stack. We then introduce file-system crash consistency, why crash-consistency bugs occur, and why it is important to test file-system crash consistency.

File-System IO. File systems typically expose the POSIX API to users in the form of system calls. The POSIX API includes data operations such as `read()` and `write()`, and metadata operations such as `creat()` and `rename()`. When the user performs a write, the write is first processed by the file system, and then handed down to the block IO subsystem as a list of blocks to be written. The blocks pass through several layers before reaching the storage device. Layers such as the IO scheduler may re-order the blocks being sent to the storage device to increase performance (e.g., sorting the blocks by logical block address helps reduce seek time on hard drives). The path taken by a file-system write operation is depicted in Figure 1.

Caching in the Storage Device. Most modern storage devices have an on-board RAM cache to increase read and write performance. When the device receives a write request, it writes the data to its cache and signals completion of the request. As a result, an IO request being *completed* by the storage device does not mean it is *persistent*; if the storage device loses power before the cached data can be persisted, the data is lost. When writing cached data to the persistent media, the storage device is free to write the data in any order; typically, the storage devices like hard disk drives will try to maximize write performance by sorting the data by logical block address.

Ordering and Durability. The file system may need to ensure ordering and durability at various points. For example, the journaling protocol requires the journal transaction be persisted before

the journal commit. Similarly, when the user calls `fsync()` on a file, they expect that all data associated with the file is persistent after the `fsync()` returns. To help implement ordering and durability, storage devices expose two flags: the `FLUSH` flag and the Forced Unit Access (`FUA`) flag. If an IO request is tagged with the `FLUSH` flag, the storage device must flush its cache and all data previously written to the storage device persistent. Note that the `FLUSH` does not force the data in its own request to be persistent, only previously written data. If an IO request is tagged with the `FUA` flag, the IO request can only return when the data in that request has been made persistent. For example, in the ext4 file system, the journal commit is tagged with both the `FLUSH` tag (ensuring durability of the previously written journal transaction) and the `FUA` flag (ensuring durability of the journal commit).

Crash Consistency. A file system is crash-consistent if a number of invariants about the file-system state hold after a crash due to power loss or a kernel panic [14, 66]. Typically, these invariants include using resources only after initialization (e.g., path-names point to initialized metadata such as inodes), safely reusing resources after deletion (e.g., two files should not think they both own the same data block), and atomically performing certain operations such as renaming a file. Conventionally, crash consistency is only concerned with internal file-system integrity. A bug that loses previously persisted data would not be considered a crash-consistency bug as long as the file system remains internally consistent. In this article, we widen the definition to include data loss. Thus, if a file system loses persisted data or files after a crash, we consider it a crash-consistency bug. The Linux file-system developers agree with this wider definition of crash consistency [19, 86]. However, it is important to note that data or metadata that has not been *explicitly persisted* does not fall under our definition; file systems are allowed to lose such data in case of power loss. Finally, there is an important difference between crash-consistency bugs and file-system correctness bugs: crash-consistency bugs *do not* lead to incorrect behavior if no crash occurs.

Why Crash-Consistency Bugs Occur. The root of crash-consistency bugs is the fact that most file-system operations *only modify in-memory state*. For example, when a user creates a file, the new file exists only in memory until it is explicitly persisted via the `fsync()` call or by a background thread which periodically writes out dirty in-memory data and metadata.

Modern file systems are complex and keep a significant number of metadata-related data structures in memory. For example, btrfs organizes its metadata as B+ trees [80]. Modifications to these data structures are accumulated in memory and written to storage either on `fsync()`, or by a background thread. Developers could make two common types of mistakes while persisting these in-memory structures, which consequently lead to crash-consistency bugs. The first is neglecting to update certain fields of the data structure. For example, btrfs had a bug where the field in the file inode that determined whether it should be persisted was not updated. As a result, `fsync()` on the file became a no-op, causing data loss on a crash [41]. The second is improperly ordering data and metadata when persisting it. For example, when delayed allocation was introduced in ext4, applications that used rename to atomically update files lost data since the rename could be persisted before the file's new data [36]. Despite the fact that the errors that cause crash-consistency bugs are very different in these two cases, the fundamental problem is that some in-memory state that is required to recover correctly is not written to disk.

POSIX and File-System Guarantees. Nominally, Linux file systems implement the POSIX API, providing guarantees as laid out in the POSIX standard [25]. Unfortunately, POSIX is extremely vague. For example, under POSIX it is legal for `fsync()` to *not* make data durable [77]. Mac OSX takes advantage of this legality, and requires users to employ `fcntl(F_FULLFSYNC)` to make data durable [3]. As a result, file systems often offer guarantees above and beyond what is required by

```
1 create foo
2 link foo bar
3 sync
4 unlink bar
5 create bar
6 fsync bar
7 CRASH!
```

Fig. 2. **Example crash-consistency bug.** The figure shows the workload to expose a crash-consistency bug that was reported in the btrfs file system in Feb. 2018 [57]. The bug causes the file system to become un-mountable.

POSIX. For example, on ext4, persisting a new file will also persist its directory entry. Unfortunately, these guarantees vary across different file systems, so we contacted the developers of each file system to ensure we are testing the guarantees that they seek to provide.

Example of a Crash-Consistency Bug. Figure 2 shows a crash-consistency bug in btrfs that causes the file system to become un-mountable (unavailable) after the crash. Resolving the bug requires file-system repair using `btrfs-check`; for lay users, this requires guidance of the developers [10]. This bug occurs on btrfs because the unlink affects two different data structures which become out of sync if there is a crash. On recovery, btrfs tries to unlink `bar` twice, producing an error.

Why Testing Crash Consistency is Important. File-system researchers are developing new crash-consistency techniques [17, 18, 75] and designing new file systems that increase performance [1, 7, 32, 35, 79, 83, 99, 100]. Meanwhile, Linux file systems such as btrfs include a number of optimizations that affect the ordering of IO requests, and hence, crash consistency. However, crash consistency is subtle and hard to get right, and a mistake could lead to silent data corruption and data loss. Thus, changes affecting crash consistency should be carefully tested.

State of Crash-Consistency Testing Today. `xfstests` [23] is a regression test suite to check file-system correctness, with a small proportion (5%) of crash-consistency tests. These tests are aimed at avoiding the recurrence of the same bug over time, but do not generalize to identifying variants of the bug. Additionally, each of these test cases requires the developer to write a checker describing the correct behavior of the file system after a crash. Given the infinite space of workloads, it is extremely hard to handcraft workloads that could reveal bugs. These factors make `xfstests` insufficient to identify *new* crash-consistency bugs.

3 STUDYING CRASH-CONSISTENCY BUGS

We present an analysis of 26 unique crash-consistency bugs reported by users over the last 5 years on widely used Linux file systems [88] (Section 11.1). We find these bugs either by examining mailing list messages or looking at the crash-consistency tests in the `xfstests` regression test suite. Few of the crash-consistency tests in `xfstests` link to the bugs that resulted in the test being written.

Due to the nature of crash-consistency bugs (all in-memory information is lost upon crash), it is hard to tie them to a specific workload. As a result, the number of reported bugs is low. We believe there are many crash-consistency bugs that go unreported in the wild (Section 11.2).

Table 1. Analyzing Crash-Consistency Bugs

Kernel Version	# bugs
3.12	3
3.13	9
3.16	1
4.1.1	2
4.4	9
4.15	3
4.16	1
Total	28

Consequence	# bugs
Corruption	19
Data Inconsistency	6
Un-mountable file system	3
Total	28

File System	# bugs
ext4	2
F2FS	2
btrfs	24
Total	28

# of ops required	# bugs
1	3
2	14
3	9
Total	26

The tables break down the 26 unique crash-consistency bugs reported over the last 5 years (since 2013) by different criteria. Two bugs were reported on two different file systems, leading to a total of 28 bugs.

Table 2. Examples of Crash-Consistency Bugs

Bug #	File System	Consequence	# of ops	Ops involved (excluding persistence operations)
1	btrfs	Directory un-removable	2	`creat(A/x)`, `creat(A/y)`
2	btrfs	Persisted data lost	2	`pwrite(x)`, `link(x,y)`
3	btrfs	Directory un-removable	3	`link(x,A/x)`, `link(x,A/y)`, `unlink(A/y)`
4	F2FS	Persisted file disappears	3	`pwrite(x)`, `rename(x,y)`, `pwrite(x)`
5	ext4	Persisted data lost	2	`pwrite(x)`, `direct_write(x)`

The table shows some of the crash-consistency bugs reported in the last 5 years. The bugs have severe consequences, ranging from losing user data to making directories un-removable.

We analyze the bugs based on consequence, kernel version, file system, and the number of file-system operations required to reproduce them. There are 26 unique bugs spread across ext4, F2FS, and btrfs. Each unique bug requires a unique set of file-system operations to reproduce. Two bugs occur on two file systems (F2FS and ext4, F2FS and btrfs), leading to a total of 28 bugs.

Table 1 presents some statistics about the crash-consistency bugs. The table presents the kernel version in which the bug was reported. If the bug report did not include a version, it presents the latest kernel version in which B^3 could reproduce the bug (the two bugs that B^3 could not reproduce appear in kernel 3.13). The bugs have severe consequences, ranging from file-system corruption to the file system becoming un-mountable. The four most common file-system operations involved in crash-consistency bugs were `write()`, `link()`, `unlink()`, and `rename()`. Most reported bugs resulted from either reusing filenames in multiple file-system operations or write operations to overlapping file regions. Most reported bugs could be reproduced with three or fewer file-system operations.

Examples. Table 2 showcases a few of the crash-consistency bugs. Bug #1 [39] involves creating two files in a directory and persisting only one of them. btrfs log recovery incorrectly counts the directory size, making the directory un-removable thereafter. Bug #2 [43] involves creating a hard link to an already existing file. A crash results in btrfs recovering the file with a size 0, thereby making its data inaccessible. A similar bug (#5 [28]) manifests in ext4 in the direct write path, where the write succeeds and blocks are allocated, but the file size is incorrectly updated to be zero, leading to data loss.

Complexity Leads to Bugs. The ext4 file system has undergone more than 15 years of development, and, as a result, has only two bugs. The btrfs and F2FS file systems are more recent: btrfs

was introduced in 2007, while F2FS was introduced in 2012. In particular, btrfs is an extremely complex file system that provides features such as snapshots, cloning, out-of-band deduplication, and compression. btrfs maintains its metadata (such as inodes and bitmaps) in the form of various copy-on-write B+ trees. This makes achieving crash consistency tricky, as the updates have to be propagated to several trees. Thus, it is not surprising that most reported crash-consistency bugs occurred in btrfs. As file systems become more complex in the future, we expect to see a corresponding increase in crash-consistency bugs.

Crash-Consistency Bugs are Hard to Find. Despite the fact that the file systems we examined were widely used, some bugs have remained hidden in them for years. For example, btrfs had a crash-consistency bug that was only discovered *7* years after it was introduced. The bug was caused by incorrectly processing a hard link in btrfs's data structures. When a hard link is added, the directory entry is added to one data structure, while the inode is added to another data structure. When a crash occurred, only one of these data structures would be correctly recovered, resulting in the directory containing the hard link becoming un-removable [46]. This bug was present since the log tree was added in 2008; however, the bug was only discovered in 2015.

Systematic Testing is Required. Once the hard link bug in btrfs was discovered, the btrfs developers quickly fixed it. However, they only fixed one code path that could lead to the bug. The same bug could be triggered in another code path, a fact that was only discovered *4 months* after the original bug was reported. While the original bug workload required creating hard links and calling `fsync()` on the original file and parent directory, this one required calling `fsync()` on a sibling in the directory where the hard link was created [47]. Systematic testing of the file system would have revealed that the bug could be triggered via an alternate code path.

Small Workloads can Reveal Bugs on an Empty File System. Most of the reported bugs do not require a special file-system image or a large number of file-system operations to reproduce. 24 out of the 26 reported bugs require three or fewer core file-system operations to reproduce on an empty file system. This count is low because we do not count *dependent* operations: for example, a file has to exist before being renamed and a directory has to exist before a file can be created inside it. Such dependent operations can be *inferred* given the core file-system operations. Of the remaining two bugs, one required a special command (`dropcaches`) to be run during the workload for the bug to manifest. The other bug required a specific setup: 3,000 hard links had to already exist (forcing an external reflink) for the bug to manifest.

Reported Bugs Involve a Crash After Persistence. All reported bugs involved a crash right after a persistence point: a call to `fsync()`, `fdatasync()`, or the global `sync` command. These commands are important because file-system operations only modify in-memory metadata and data by default. Only persistence points reliably change the file-system state on storage. Therefore, unless a file or directory has been persisted, it cannot be expected to survive a crash. While crashes could technically occur at any point, a user cannot complain if a file that has not been persisted goes missing after a crash. Thus, every crash-consistency bug involves *persisted data or metadata* that are affected by the bug after a crash, and a workload that does not have a persistence point cannot lead to a reproducible crash-consistency bug. This also points to an effective way to find crash-consistency bugs: perform a sequence of file-system operations, change on-storage file-system state with `fsync()` or similar calls, crash, and then check files and directories that were previously persisted.

4 B^3: BOUNDED BLACK-BOX CRASH TESTING

Based on the insights from our study of crash-consistency bugs, we introduce a new approach to testing file-system crash consistency: *Bounded Black-Box crash testing* (B^3). B^3 is a black-box

testing approach built upon the insight that most reported crash-consistency bugs can be found by systematically testing small sequences of file-system operations on a new file system. B^3 exercises the file system through its system-call API, and observes the file-system behavior via read and write IO. As a result, B^3 does not require annotating or modifying file-system source code.

4.1 Overview

B^3 generates sequences of file-system operations, called *workloads*. Since the space of possible workloads is infinite, B^3 *bounds* the space of workloads using insights from the study. Within the determined bounds, B^3 exhaustively generates and tests all possible workloads. Each workload is tested by *simulating* a crash after each persistence point, and checking if the file system recovers to a correct state. B^3 performs fine-grained correctness checks on the recovered file-system state; only files and directories that were explicitly persisted are checked. B^3 checks for both data and metadata (size, link count, and block count) consistency for files and directories.

Crash Points. The main insight from the study that makes an approach like B^3 feasible is the choice of crash points; a crash is simulated *only* after each persistence point in the workload instead of in the middle of file-system operations. This design choice was motivated by two factors. First, file-system guarantees are undefined if a crash occurs in the middle of a file-system operation; only files and directories that were previously successfully persisted need to survive the crash. File-system developers are overloaded, and bugs involving data or metadata that has not been explicitly persisted is given low priority (and sometimes not acknowledged as a bug). Second, if we crash in the middle of an operation, there are a number of correct states the file system could recover to. If a file-system operation translates to n block IO requests, there could be 2^n different on-disk crash states if we crashed anywhere during the operation. Restricting crashes to occur after persistence points bounds this space linearly in the number of operations comprising the workload. The small set of crash points and correct states makes automated testing easier. Our choice of crash points naturally leads to bugs where persisted data and metadata are corrupted or missing and file-system developers are strongly motivated to fix such bugs.

4.2 Bounds Used by B^3

Based on our study of crash-consistency bugs, B^3 bounds the space of possible workloads in several ways:

(1) **Number of Operations.** B^3 bounds the number of file-system operations (termed the *sequence length*) in the workload. A `seq-X` workload has X core file-system operations in it, not counting dependent operations such as creating a file before renaming it.
(2) **Files and Directories in Workload.** We observe that in the reported bugs, errors result from the *reuse* of a small set of files for metadata operations. Thus, B^3 restricts workloads to use few files per directory, and a low directory depth. This restriction automatically reduces the inputs for metadata-related operations such as `rename()`.
(3) **Data Operations.** The study also indicated that bugs related to data inconsistency mainly occur due to writes to *overlapping* file ranges. In most cases, the bugs are not dependent on the exact offset and length used in the writes, but on the interaction between the overlapping regions from writes. The study indicates that a broad classification of writes such as appends to the end of a file, overwrites to overlapping regions of a file, and so on, is sufficient to find crash-consistency bugs.
(4) **Initial File-System State.** Most of the bugs analyzed in the study did not require a specific initial file-system state (or a large file system) to be revealed. Moreover, most of the

studied bugs could be reproduced starting from the *same, small* file-system image. Therefore, B^3 can test all workloads starting from the same initial file-system state.

4.3 Fine-Grained Correctness Checking

B^3 uses fine-grained correctness checks to validate the data and metadata of persisted files and directories in each crash state. Since `fsck` is both time-consuming to run and can miss data loss/corruption bugs, it is not a suitable checker for B^3.

4.4 Limitations

The B^3 approach has a number of limitations:

(1) B^3 does not make any guarantees about finding *all* crash-consistency bugs. It is sound but incomplete. However, because B^3 tests exhaustively, if the workload that triggers the bug falls within the constrained workload space, B^3 will find it. Therefore, the effectiveness of B^3 depends upon the bounds chosen and the number of workloads tested.
(2) B^3 focuses on a specific class of bugs. It does not simulate a crash in the middle of a file-system operation and it does not re-order IO requests to create different crash states. The implicit assumption is that the core crash-consistency mechanism, such as journaling [78] or copy-on-write [30, 81], is working correctly. Instead, we assume that it is the rest of the file system that has bugs. The crash-consistency bug study indicates this assumption is reasonable. We explore crashing in the middle of file-system operations in Section 7, which did not result in finding any new bugs.
(3) B^3 focuses on workloads where files and directories are explicitly persisted. If we created a file, waited 1 hour, then crashed, and found that the file was gone after the file-system recovered, this would also be a crash-consistency bug. However, B^3 does not explore such workloads as they take a significant amount of time to run and are not easily reproduced in a deterministic fashion.
(4) Due to its black-box nature, B^3 cannot pinpoint the exact lines of code that result in the observed bug. Once a bug has been revealed by B^3, finding the root cause requires further investigation. However, B^3 aids in investigating the root cause of the bug since it provides a way to reproduce the bug in a deterministic fashion.

Despite its shortcomings, we believe B^3 is a useful addition to the arsenal of techniques for testing file-system crash consistency. The true strengths of B^3 lie in its systematic nature and the fact that it does not require any changes to existing systems. Therefore, it is ideal for complex and widely used file systems written in low-level languages like C, where stronger approaches like verification cannot be easily used.

5 CRASHMONKEY AND ACE

We realize the B^3 approach by building two tools, CrashMonkey and Ace. As shown in Figure 3, CrashMonkey is responsible for simulating crashes at different points of a given workload and testing if the file system recovers correctly after each simulated crash, while the Automatic Crash Explorer (Ace) is responsible for exhaustively generating workloads in a bounded space.

5.1 CrashMonkey

CrashMonkey uses record-and-replay techniques to *simulate* a crash in the middle of the workload and test if the file system recovers to a correct state after the crash. For maximum portability, CrashMonkey treats the file system as a black box, only requiring that the file system implement the POSIX API.

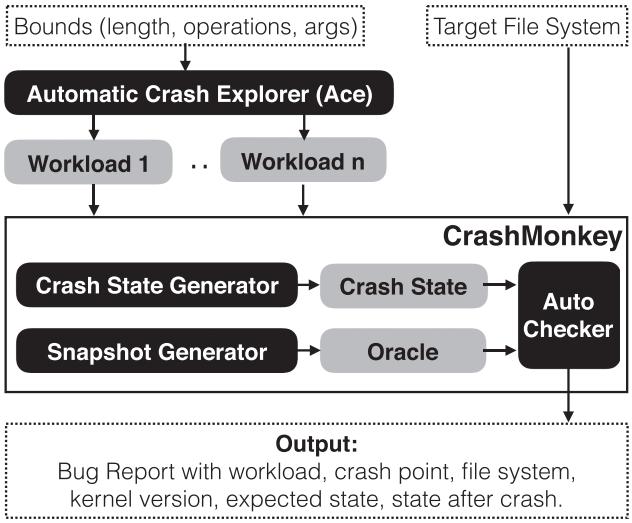

Fig. 3. **System architecture.** Given bounds for exploration, Ace generates a set of workloads. Each workload is then fed to CrashMonkey, which generates a set of crash states and corresponding oracles. The AutoChecker compares persisted files in each oracle/crash state pair; a mismatch indicates a bug.

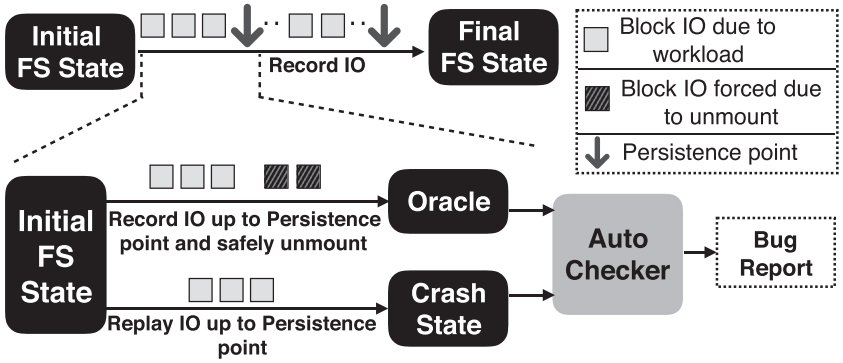

Fig. 4. **CrashMonkey operation.** CrashMonkey first records the block IO requests that the workload translates to, capturing reference images called oracles after each persistence point. CrashMonkey then generates crash states by replaying the recorded IO and tests for consistency against the corresponding oracle.

Overview. CrashMonkey operates in three phases as shown in Figure 4. In the first phase, CrashMonkey profiles the workload by collecting information about all file-system operations and IO requests made during the workload. The second phase replays IO requests until a persistence point to create a *crash state*. The crash state represents the state of storage if the system had crashed after a persistence operation completed. CrashMonkey then mounts the file system in the crash state and allows the file system to perform recovery. At each persistence point, CrashMonkey also captures a reference file-system image, termed the *oracle*, by safely unmounting it so the file system completes any pending operations or checkpointing. The oracle represents the expected state of the file system after a crash. We rely on the correctness of the `umount()` operation to generate an oracle image. If the `umount()` operation itself does not flush all the dirty blocks, then it is possible that we miss finding a bug. However, in all the previously reported crash-consistency bugs

we explored, we never encountered this situation. In the absence of bugs, persisted files should be the same in the oracle and the crash state after recovery. In the third phase, CrashMonkey's AutoChecker tests for correctness by comparing the persisted files and directories in the oracle with the crash state after recovery.

CrashMonkey is implemented as two kernel modules and a set of user-space utilities. The kernel modules consist of 1,300 lines of C code which can be compiled and inserted into the kernel at runtime, thus avoiding the need for long kernel re-compilations. The user-space utilities consist of 4,800 lines of C++ code. CrashMonkey's separation into kernel modules and user-space utilities allows rapid porting to a different kernel version; only the kernel modules need to be ported to the target kernel. This allowed us to port CrashMonkey to seven kernels to reproduce the bugs studied in Section 3.

Profiling Workloads. CrashMonkey profiles workloads at two levels of the storage stack: it records block IO requests, and it records system calls. It uses two kernel modules to record block IO requests and create crash states and oracles.

The first kernel module records all IO requests generated by the workload using a wrapper block device on which the target file system is mounted. The wrapper device records both data and metadata for IO requests (such as sector number, IO size, and flags). Each persistence point in the workload causes a special *checkpoint* request to be inserted into the stream of IO requests recorded. The checkpoint is simply an empty block IO request with a special flag, to correlate the completion of a persistence operation with the low-level block IO stream. All the data recorded by the wrapper device is communicated to the user-space utilities via `ioctl` calls.

The second kernel module in CrashMonkey is an in-memory, copy-on-write block device that facilitates snapshots. CrashMonkey creates a snapshot of the file system before the profiling phase begins, which represents the base disk image. CrashMonkey provides fast, writable snapshots by replaying the IO recorded during profiling on top of the base disk image to generate a crash state. Snapshots are also saved at each persistence point in the workload to create oracles. Furthermore, since the snapshots are copy-on-write, resetting a snapshot to the base image simply means dropping the modified data blocks, making it efficient.

CrashMonkey also records all `open()`, `close()`, `fsync()`, `fdatasync()`, `rename()`, `sync()`, and `msync()` calls in the workload so that when the workload does a persistence operation such as `fsync(fd)`, CrashMonkey is able to correlate `fd` with a file that was opened earlier. This allows CrashMonkey to track the set of files and directories that were explicitly persisted at any point in the workload. This information is used by CrashMonkey's AutoChecker to ensure that only files and directories explicitly persisted at a given point in the workload are compared. CrashMonkey uses its own set of functions that wrap system calls which manipulate files to record the required information.

Constructing Crash States. To create a crash state, CrashMonkey starts from the initial state of the file system (before the workload was run), and uses a utility similar to `dd` to replay all recorded IO requests from the start of the workload until the next checkpoint in the IO stream. The resultant crash state represents the state of the storage just after the persistence-related call completed on the storage device. Since the IO stream replay ends directly after the next persistence point in the stream, the generated crash point represents a file-system state that is considered uncleanly unmounted. Therefore, when the file system is mounted again, the kernel may run file-system-specific recovery code.

Automatically Testing Correctness. CrashMonkey's AutoChecker is able to test for correctness automatically because it has three key pieces of information: it knows which files were

Phase 1: Select operations	**Phase 2:** Select parameters	**Phase 3:** Add persistence points	**Phase 4:** Add dependencies

			`mkdir(A)`
			`mkdir(B)`
			`create(A/foo)`

`1 rename()`
`2 link()`
	`1 rename(A/foo,B/bar)`
	`2 link(B/bar, A/bar)`
		`1 rename(A/foo,B/bar)`
		`sync()`
		`2 link(B/bar, A/bar)`
		`fsync(A/bar)`
			`1 rename(A/foo,B/bar)`
			`sync()`
			`2 link(B/bar, A/bar)`
			`fsync(A/bar)`

Fig. 5. **Workload generation in Ace.** The figure shows the different phases involved in workload generation in Ace. Given the sequence length, Ace first selects the operations, then selects the parameters for each operation, then optionally adds persistence points after each operation, and finally satisfies file and directory dependencies for the workload. The final workload may have more operations than the original sequence length.

persisted, it has the correct data and metadata of those files in the oracle, and it has the actual data and metadata of the corresponding files in the crash state after recovery. Testing correctness is a simple matter of comparing data and metadata of persisted files in the oracle and the crash state.

CrashMonkey avoids using `fsck` because its runtime is proportional to the amount of data in the file system (not the amount of data changed) and it does not detect the loss or corruption of user data. Instead, when a crash state is re-mounted, CrashMonkey allows the file system to run its recovery mechanism, like journal replay, which is usually more lightweight than `fsck`. `fsck` is run only if the recovered file system is un-mountable. To check consistency, CrashMonkey uses its own *read* and *write* checks after recovery. The read checks used by CrashMonkey confirm that persisted files and directories are accurately recovered. The write checks test if a bug makes it impossible to modify files or directories. For example, a btrfs bug made a directory un-removable due to a stale file handle [39].

Since each file system has slightly different consistency guarantees, we reached out to developers of each file system we tested, to understand the guarantees provided by that file system. In some cases, our conversations prompted the developers to explicitly write down the persistence guarantees of their file systems for the first time [87]. During this process, we confirmed that most file systems such as ext4 and btrfs implement a stronger set of guarantees than the POSIX standard. For example, while POSIX requires an `fsync()` on both a newly created file and its parent directory to ensure the file is present after a crash, many Linux file systems do not require the `fsync()` of the parent directory. Based on the response from developers, we report automatically detected bugs that violate the guarantees each file system aims to provide.

5.2 Automatic Crash Explorer (Ace)

Ace exhaustively generates workloads satisfying the given bounds. Ace has two components, the workload synthesizer and the adapter for CrashMonkey.

Workload Synthesizer. The workload synthesizer exhaustively generates workloads within the state space defined by the user-specified bounds. The workloads generated in this stage are represented in a high-level language, similar to the one depicted in Figure 5.

CrashMonkey Adapter. A custom adapter converts the workload generated by the synthesizer into an equivalent C++ test file that CrashMonkey can work with. This adapter handles the insertion of wrapped file-system operations that CrashMonkey tracks. Additionally, it inserts a special

Table 3. Bounds Used by ACE

B^3 bound	Insight from the study	Bound chosen by ACE
Number of operations	Small workloads of two to three core operations	Maximum # of core ops in a workload is *three*
Files and directories	Re-use file and directory names	Two directories of depth 2, each with two unique files
Data operations	Coarse-grained, overlapping ranges of writes	Overwrites to start, middle, and end of file, and appends
Initial file-system state	No need of a special initial state or large image	Start with a clean file-system image of size 100MB

The table shows the specific values picked by ACE for each B^3 bound.

function-call at every persistence point, which translates to the checkpoint IO. It is easy to extend ACE to be used with other record-and-replay tools like `dm-log-writes` [4] by building custom adapters.

Table 3 shows how we used the insights from the study to assign specific values for B^3 bounds when we run ACE. Given these bounds, ACE uses a multi-phase process to generate workloads that are then fed into CRASHMONKEY. Figure 5 illustrates the four phases ACE goes through to generate a `seq-2` workload.

Phase 1: Select Operations and Generate Workloads. ACE first selects file-system operations for the given sequence length to make what we term the *skeleton*. By default, file-system operations can be repeated in the workload. The user may also supply bounds such as requiring only a subset of file-system operations be used (e.g., to focus testing on new operations). ACE then exhaustively generates workloads satisfying the given bounds. For example, if the user specified the `seq-2` workload could only contain six file-system operations, ACE will generate $6 * 6 = 36$ skeletons in phase one.

Phase 2: Select Parameters. For each skeleton generated in phase one, ACE then selects the parameters (system-call arguments) for each file-system operation. By default, ACE uses two files at the top level and two sub-directories with two files each as arguments for metadata-related operations. ACE also understands the semantics of file-system operations and exploits it to eliminate the generation of *symmetrical* workloads. For example, consider two operations `link(foo, bar)` and `link(bar, foo)`. The idea is to link two files within the same directory, but the order of file names chosen does not matter. In this example, one of the workloads would be discarded, thus reducing the total number of workloads to be tested for the sequence.

For data operations, ACE chooses between whether a write is an overwrite at the beginning, middle, or end of the file or simply an append operation. Furthermore, since our study showed that crash-consistency bugs occur when data operations overlap, ACE tries to overlap data operations in phase two.

Each skeleton generated in phase one can lead to multiple workloads (based on different parameters) in phase two. However, at the end of this phase, each generated workload has a sequence of file-system operations with all arguments identified.

Phase 3: Add Persistence Points. ACE optionally adds a persistence point after each file-system operation in the workload, but ACE does not require every operation to be followed by a persistence point. However, ACE ensures that the last operation in a workload is always followed by a persistence point so that it is not truncated to a workload of lower sequence length. The file or directory to be persisted in each call is selected from the same set of files and directories used

by phase two, and, for each workload generated by phase two, phase three can generate multiple workloads by adding persistence points after different sets of file-system operations.

Phase 4: Add Dependencies. Finally, ACE satisfies various dependencies to ensure the workload can execute on a POSIX file system. For example, a file has to exist before being renamed or written to. Similarly, directories have to be created if any operations on their files are involved. Figure 5 shows how `A`, `B`, and `A/foo` are created as dependencies in the workload. As a result, a `seq-2` workload can have more than two file-system operations in the final workloads. At the end of this phase, ACE compiles each workload from the high-level language into a C++ program that can be passed to CrashMonkey.

Implementation. ACE consists of 2,500 lines of Python code, and currently supports 14 file-system operations. All bugs analyzed in our study used one of these 14 file-system operations. It is straightforward to expand ACE to support more operations.

Running Ace with Relaxed Bounds. It is easy to relax the bounds used by ACE to generate more workloads; this comes at the cost of computational time used to test the extra workloads. Care should be taken when relaxing the bounds, since the number of workloads increases at a rapid rate. For example, ACE generates about 1.5M workloads with three core file-system operations. Relaxing the default bound on the set of files and directories to add one additional nested directory, increases the number of workloads generated to 3.7M. This simple change results in 2.5× more workloads. Note that increasing the number file-system operations in the workload leads to an increase in the number of phase-1 skeletons generated, and adding more files to the argument set increases the number of phase-2 workloads that can be created. Therefore, the workload space must be carefully expanded.

5.3 Testing and Bug Analysis

Testing Strategy. Given a target file system, we first exhaustively generate `seq-1` workloads and test them using CrashMonkey. We then proceed to `seq-2`, and then `seq-3` workloads. By generating and testing workloads in this order, CrashMonkey only needs to simulate a crash at one point per workload. For example, even if a `seq-2` workload has two persistence points, crashing after the first persistence point would be equivalent to an already-explored `seq-1` workload.

Analyzing Bug Reports. One of the challenges with a black-box approach like B^3 is that a single bug could result in many different workloads failing correctness tests. We present two cases of multiple test failures in workloads, and how we mitigate them.

First, workloads in different sequences can fail because of the same bug. Our testing strategy is designed to mitigate this: if a bug causes incorrect behavior with a single file-system operation, it should be caught by a `seq-1` workload. Therefore, if we catch a bug only in a `seq-2` workload, it implies the bug results from the interaction of the two file-system operations. Ideally, we would run `seq-1`, report any bugs, and apply bug-fix patches given by developers before running `seq-2`. However, for quicker testing, ACE maintains a database of all previously found bugs which includes the core file-system operations that produced each bug and the consequence of the bug. For all new bugs reports generated by CrashMonkey and ACE, it first compares the workload and the consequence with the database of known bugs. If there is a match, ACE does not report the bug to the user.

Second, similar workloads in the same sequence could fail correctness tests due to the same bug. For efficient analysis, we group together bug reports by the consequence (e.g., file missing), and the skeleton (the sequence of core file-system operations that comprise the workload) that triggered the bug, as shown in Figure 6. Using the skeleton instead of the fully fleshed-out workload allows

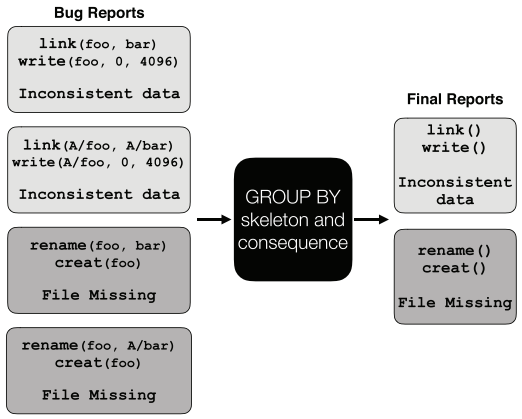

Fig. 6. **Post-processing.** The figure shows how generated bug reports are processed to eliminate duplicates.

us to identify similar bugs. For example, the bug that causes appended data to be lost will repeat four times, once with each of the files in our file set. We can group these bug reports together and only inspect one bug report from each group. After verifying each bug, we report it to developers.

6 EVALUATION

We evaluate the utility and performance of the B^3 approach by answering the following questions:

- —Do CrashMonkey and Ace find known bugs and new bugs in Linux file systems in a reasonable period of time? (Section 6.2)
- —What is the performance of CrashMonkey? (Section 6.3)
- —What is the performance of Ace? (Section 6.4)
- —How much memory and CPU does CrashMonkey consume? (Section 6.5)

6.1 Experimental Setup

B^3 requires testing a large number of workloads in a systematic manner. To accomplish this testing, we deploy CrashMonkey on Chameleon Cloud [38], an experimental testbed for large-scale computation.

We employ a cluster of 65 nodes on Chameleon Cloud. Each node has 24 cores, 128GB RAM, and 250GB HDD. We install 12 VirtualBox virtual machines running Ubuntu 16.04 LTS on each node, each with 2GB RAM and 10GB storage. Each virtual machine runs one instance of CrashMonkey. Thus, we have a total of 780 virtual machines testing workloads with CrashMonkey in parallel. Based on the number of virtual machines that could be reliably supported per node, we limited the total to 780.

On a local server, we generate the workloads with Ace and divide them into sets of workloads to be tested on each virtual machine. We then copy the workloads over the network to each physical Chameleon node, and, from each node, copy them to the virtual machines.

6.2 Bug Finding

Determining Workloads. Our goal was to test whether the B^3 approach was useful and practical, not to exhaustively find every crash-consistency bug. Therefore, we wanted to limit the computational time spent on testing to a few days. Thus, we needed to determine what workloads to test with our computational budget.

Table 4. Workloads Tested

Sequence type	File-system operations tested	# of workloads	Runtime (minutes)
seq-1	creat, mkdir, falloc, buffered write, mmap,	300	1
seq-2	link direct-IO write, unlink, rmdir, setxattr removexattr, remove, unlink, truncate	254K	215
seq-3-data	buffered write, mmap, direct-IO write, falloc	120K	102
seq-3-metadata	buffered write, link, unlink, rename	1.5M	1,274
seq-3-nested	link, rename	1.5M	1,274
Total		3.37M	2,866

The table shows the number of workloads tested in each set, along with the time taken to test these workloads in parallel on 65 physical machines and the file-system operations tested in each category. Overall, we tested 3.37 million workloads in 2 days, reproducing 24 known bugs and finding 10 new crash-consistency bugs.

Our study of crash-consistency bugs indicated that it would be useful to test small workloads of length one, two, and three. However, we estimated that testing all 25 million possible workloads of length three was infeasible within our target time-frame. We had to further restrict the set of workloads that we tested. We used our study to guide us in this task. At a minimum, we wanted to select bounds that would generate the workloads that reproduced the reported bugs. Using this as a guideline, we came up with a set of workloads that was broad enough to reproduce existing bugs (and potentially find new bugs), but small enough that we could test the workloads in a few days on our research cluster.

Workloads. We test workloads of length one (`seq-1`), two (`seq-2`), and three (`seq-3`). We further separate workloads of length three into three groups: one focusing on data operations (`seq-3-data`), one focusing on metadata operations (`seq-3-metadata`), and one focusing on metadata operations involving a file at depth three (`seq-3-nested`) (by default, we use depth two).

The `seq-1` and `seq-2` workloads use a set of 14 file-system operations. For `seq-3` workloads, we narrow down the list of operations, based on what category the workload is in. The complete list of file-system operations tested in each category is shown in Table 4.

Testing Strategy. We tested `seq-1` and `seq-2` workloads on ext4, xfs, F2FS, and btrfs, but did not find any new bugs in ext4 or xfs. We additionally tested the `seq-1` workloads on two verified file systems, FSCQ and Yxv6. We focused on F2FS and btrfs for the larger `seq-3` workloads. In total, we spent 48 hours testing all 3.37 million workloads per file system on the 65-node research cluster described earlier. Table 4 presents the number of workloads in each set, and the time taken to test them (for each file system). All the tests are run only on 4.16 kernel. To reproduce reported bugs, we employ the following strategy. We encode the workload that triggers previously reported bugs in Ace. In the course of workload generation, when Ace generates a workload identical to the encoded one, it is added to a list. This list of workloads is run on the kernel versions reported in Table 1, to validate that the workload produced by Ace can indeed reproduce the bug.

Cost of Computation. We believe the amount of computational effort required to find crash-consistency bugs with CrashMonkey and Ace is reasonable. For example, if we were to rent 780 `t2.small` instances on Amazon to run Ace and CrashMonkey for 48 hours, at the current rate of $0.023 per hour for on-demand instances [2], it would cost $780 * 48 * 0.023 = \$861.12$. For the complete 25M workload set, the cost of computation would go up by 7.5×, totaling $6.4K. Thus, we can test each file system for less than $7K. Alternatively, a company can provision physical nodes to run the tests; we believe this would not be hard for a large company.

Table 5. Newly Discovered Bugs

Bug #	File System	Consequence	# of ops	Bug present since
1	btrfs	Rename atomicity broken (file disappears)	3	2014
2	btrfs	Rename atomicity broken (file in both locations)	3	2018
3	btrfs	Directory not persisted by fsync*	3	2014
4	btrfs	Rename not persisted by fsync	3	2014
5	btrfs	Hard links not persisted by fsync*	2	2014
6	btrfs	Directory entry missing after fsync on directory	2	2014
7	btrfs	Fsync on file does not persist all its paths*	1	2014
8	btrfs	Allocated blocks lost after fsync*	1	2014
9	F2FS	File recovers to incorrect size*	1	2015
10	F2FS	Persisted file disappears*	2	2016
11	FSCQ	File data loss*	1	2018

The table shows the new bugs found by CrashMonkey and Ace. The bugs have severe consequences, ranging from losing allocated bocks to entire files and directories disappearing. The bugs have been present for several years in the kernel, showing the need for systematic testing. Note that even workloads with single file-system operation have resulted in bugs. Developers have submitted a patch for bugs marked with*.

Results. CrashMonkey and Ace found 10 **new** crash-consistency bugs [89] in btrfs and F2FS and 1 new bug in FSCQ, in addition to reproducing 24 out of 26 bugs reported over the past 5 years. We studied the bug reports for the new bugs to ensure they were unique and not different manifestations of the same underlying bug. We verified each unique bug triggers a different code path in the kernel, indicating the root cause of each bug is not the same underlying code.

All new bugs were reported to file-system developers and acknowledged [15, 16, 71, 72]. Developers have submitted patches for five bugs [11, 55, 59, 97, 98], and are working on patches for the others [58]. Table 5 presents the new bugs discovered by CrashMonkey and Ace. We make several observations based on these results.

The Discovered Bugs have Severe Consequences. The newly discovered bugs result in either data loss (due to missing files or directories) or file-system corruption. More importantly, the missing files and directories have been *explicitly persisted* with an `fsync()` call and thus should survive crashes.

Small Workloads are Sufficient to Reveal New Bugs. One might expect only workloads with two or more file-system operations to expose bugs. However, the results show that even workloads consisting of a single file-system operation, if tested systematically, can reveal bugs. For example, three bugs were found by `seq-1` workloads, where CrashMonkey and Ace only tested 300 workloads in a systematic fashion. Interestingly, variants of these bugs have been patched previously, and it was sufficient to simply change parameters to file-system operations to trigger the same bug through a different code-path.

An F2FS bug found by CrashMonkey and Ace is a good example of finding variants of previously patched bugs. The previously patched bug manifested when `fallocate()` was used with the `KEEP_SIZE` flag; this allocates blocks to a file but does not increase the file size. By calling `fallocate()` with the `KEEP_SIZE` flag, developers found that F2FS only checked the file size to see if a file had been updated. Thus, `fdatasync()` on the file would have no result. After a crash, the file recovered to an incorrect size, thereby not respecting the `KEEP_SIZE` flag. This bug was patched in Nov. 2017 [96]; however, the `fallocate()` system call has several more flags like `ZERO_RANGE`, `PUNCH_HOLE`, and so on, and developers failed to systematically test all possible parameter options of the system call. Therefore, our tools identified and reported that the same bug can appear when

ZERO_RANGE is used. Though this bug was recently patched by developers, it provides more evidence that the state of crash-consistency testing today is insufficient, and that systematic testing is required.

Crash-Consistency Bugs are Hard to Find Manually. CrashMonkey and Ace found eight new bugs in btrfs in kernel 4.16. Interestingly, seven of these bugs have been present since kernel 3.13, which was released in 2014. The ability of our tools to find *four-year-old* crash-consistency bugs within 2 days of testing on a research cluster of modest size speaks to both the difficulty of manually finding these bugs, and the power of systematic approaches like B^3.

Broken Rename Atomicity Bug. Ace generated several workloads that broke the rename atomicity of btrfs. The workloads consist of first creating and persisting a file such as A/bar. Next, the workload creates another file B/bar, and tries to replace the original file, A/bar, with the new file. The expectation is that we are able to read either the original file, A/bar, or the new file, B/bar. However, btrfs can lose both A/bar and B/bar if it crashes at the wrong time. While losing rename atomicity is bad, the most interesting part of this bug is that fsync() must be called on an un-related sibling file, like A/foo, before the crash. This shows that workloads revealing crash-consistency bugs are hard for a developer to find manually since they do not always involve obvious sequences of operations.

Crash-Consistency Bugs in Verified File Systems. CrashMonkey and Ace found a crash-consistency bug in FSCQ [13] that led to data loss in spite of persisting the file using fdatasync(). The developers have acknowledged and patched the bug [11]. The origin of this bug can be tracked down to an optimization introduced in the C-Haskell binding in FSCQ, which is unverified code.

6.3 CrashMonkey Performance

CrashMonkey has three phases of operation: profiling the given workload, constructing crash states, and testing crash-consistency. Given a workload, the end-to-end latency to generate a bug report is 4.6 seconds. The main bottleneck is the kernel itself: mounting a file system requires up-to a second of delay (if CrashMonkey checks file-system state earlier, it sometimes gets an error). Similarly, once the workload is done, we also wait for 2 seconds to ensure the storage subsystem has processed the writes, and that we can unmount the file system without affecting the writes. These delays account for 84% of the time spent profiling.

After profiling, constructing crash states is relatively fast: CrashMonkey only requires 20ms to construct each crash state. Furthermore, since CrashMonkey uses fine-grained correctness tests, checking crash consistency with both read and write tests takes only 20ms. A more detailed break-down of time taken by various phases of CrashMonkey is as shown in Table 6.

We further optimize the running time of CrashMonkey based on few observations of the workload pattern. First, the 2-second writeback delay is unnecessary, as all our workloads end with a persistence operation. To create an oracle, we safely unmount the file system, thereby forcing these pending writes to disk. The writeback delay is only necessary for workloads that do not terminate at a persistence operation. To ensure that we record the block IO requests in the order they were issued, it is necessary to wait until the writeback delay in such cases. This optimization of disabling writeback delay wait, saves 2 seconds of running time per CrashMonkey test. Second, we see that the mount (or unmount) delay is not absolute. Hence, the 1-second mount (or unmount) delay is eliminated by successively retrying after every 500-μs. These optimizations bring down the running time of CrashMonkey from 4.6 seconds to 1.5 seconds.

Table 6. CrashMonkey Performance

Stages	# Time taken (ms)
Profiling the workload	
Formatting and mounting the device	100
Mount delay	1,000
Insert kernel modules	378
Snapshot	56
Writeback delay	2,000
Unmounting the device	7
Sub Total	3,541
Constructing a crash state	21
Testing consistency	
Mounting disks	1,009
Read checks	16
Write checks	3
Sub Total	1,028
Total	4,590

The table shows the time spent in each stage of CrashMonkey. The end-to-end latency to output the bug report is 4.6 seconds.

6.4 Ace Performance

Ace generated all the workloads that were tested (3.37M) in 374 minutes (≈150 workloads generated per second). Despite this high cost, it is important to note that generating workloads is a one-time cost. Once the workloads are generated, CrashMonkey can test these workloads on different file systems without any reconfiguration.

Deploying these workloads to the 780 virtual machines on Chameleon took 237 minutes: 34 minutes to group the workloads by virtual machines, 199 minutes to copy workloads to the Chameleon nodes, and 4 minutes to copy workloads to the virtual machines on each node.

These numbers reflect the time taken for a single local server to generate and push the workloads to Chameleon. By utilizing more servers and employing a more sophisticated strategy for generating workloads, we could reduce the time required to generate and push workloads.

6.5 Resource Consumption

The total memory consumption by CrashMonkey averaged across 10 randomly chosen workloads and the three sequence lengths are 20.12MB. The low memory consumption results from the copy-on-write nature of the wrapper block device. Since Ace's workloads typically modify small amounts of data or metadata, the modified pages are few in number, resulting in low memory consumption. Furthermore, CrashMonkey uses persistent storage only for storing the workloads (480KB per workload). Finally, the CPU consumption of CrashMonkey, as reported by `top`, was negligible (less than 1%).

7 SIMULATING CRASHES IN THE MIDDLE OF FILE-SYSTEM OPERATIONS

CrashMonkey and Ace focus on a specific class of bugs that arise due to crashes after persistence points. We relax this assumption and extend CrashMonkey to find crash-consistency bugs that could arise due to crashes in the middle of file-system operations. For example, the IO requests could be re-ordered by the storage device, in the absence of a `FLUSH` or `FUA` flag. Suppose the file system has a bug in the journaling mechanism where the journal commit is not followed by either

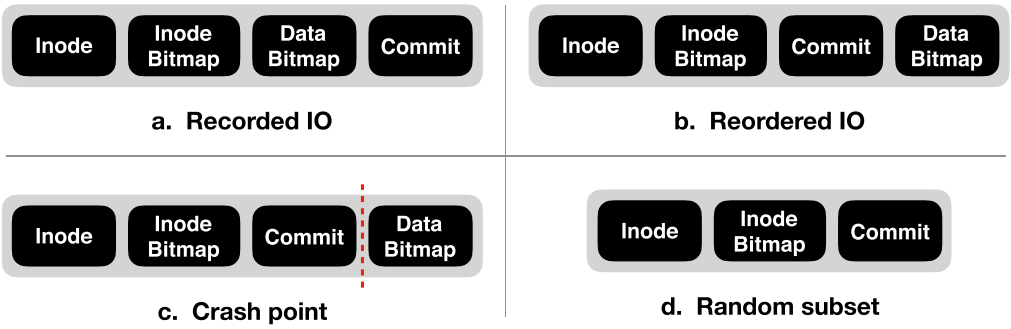

Fig. 7. **Bug in the journaling mechanism.** If the journal entries are logged without a terminating FLUSH or FUA flag, they could be reordered at the volatile storage device cache on a crash. The resulting crash state could be any random subset of the logged IOs, thereby resulting in file system inconsistency.

of these flags as shown in Figure 7(a). If a crash occurs in the middle of a journal update, these IO requests could be re-ordered or dropped. Figure 7(b) depicts a scenario where the commit block and data bitmap are re-ordered at the storage device cache. If a crash now occurs after the commit block is flushed to disk as shown in Figure 7(c), the resultant crash state shown in Figure 7(d) leads to a file-system inconsistency. In this case, the inode claims that the data blocks are allocated, while according to the data bitmap, they are not. Additionally, the inode points to garbage data. Note that the crash state in Figure 7(d) is simply a randomly picked subset of the initial recorded IO in Figure 7(a).

We extend CrashMonkey to find bugs of this kind, that could arise due to an incorrect core journaling mechanism of file systems and to simulate the behavior of the storage device that could re-order or drop IO requests. CrashMonkey supports a `subset-replay` mode, where a random subset of the workload is replayed and the user is expected to manually provide correctness tests. CrashMonkey does not support automatic correctness checks in this mode, because there are many correct states to which the file system could recover when a crash occurs in the middle of a system call. For example, if a crash occurs during a write system call, the file could either be empty, or in any partially written state when it recovers. File system guarantees are not concrete in such scenarios, with no single correct state to which the file system is expected to recover. Therefore, CrashMonkey requires that the user specify a list of correctness checks to be performed after the execution of workloads in this mode. Since there are numerous correct states to which the file system could recover, manually writing these tests could be tedious.

The `subset-replay` mode of CrashMonkey also works in the same three phases described in Section 5.1. In the first phase, CrashMonkey profiles the workload and collects the sequence of IO requests issued. The second phase replays IO requests to create a *crash state*, by selecting a random subset of the requests. The crash state represents the state of storage if the system had crashed after a subset of the disk requests completed. In the third phase, CrashMonkey tests for correctness by running the user-provided correctness tests. These user tests must implement a specific interface, and are provided information about how far into the workload the generated crash state is. When the file system is mounted in the crash state, it is allowed to perform any journal recovery, following which the user tests are run to check the existence of a file, directory, or their contents. In these tests, the user is responsible for determining if the file system is in a consistent state.

Generating Random Subsets of IOs. In the `subset-replay` mode, CrashMonkey generates crash states by breaking the set of recorded disk IOs into *epochs*, where each epoch consists of a

set of IO requests up to and including a request with the FLUSH or FUA flag set on it. Both flags represent a sort of barrier operation for the storage device because the FLUSH flag tells the storage device to flush its on-board cache to persistent storage while the FUA flag tells the storage device not to mark the current request complete until it is persisted. Once the workload has been divided into epochs, a randomly chosen number of epochs are replayed in their entirety, and then a subset of the following epoch is replayed to generate the crash state. In the final partial epoch, requests are replayed in the order they were recorded in, but some of the requests may be dropped, simulating a storage device that crashed while some data was stored in a volatile cache. If there are two writes to the same logical block address, the `subset-replay` mode is careful not to re-order a later write before the earlier one, as the storage device cache would merge these requests.

CrashMonkey can also split the recorded workload into what we call *soft epochs*, which use an additional criteria to split IO requests into epochs. While a normal epoch will only end if a request with a FLUSH or FUA flag is seen, soft epochs can also end if some amount of time elapses (default of 2.5 seconds) between consecutive requests. This feature aims to more faithfully represent storage devices since it is likely that a storage device would move data from its volatile cache to persistent storage if no new requests are coming in.

Dealing with Large Number of Crash States. The `subset-replay` mode of CrashMonkey randomly picks the number of epochs to replay, and the number of requests to be included in the final partial epoch in which the crash occurs. This results in a large number of possible crash states for any given workload. For a workload with several epochs and tens of IOs per epoch, it is rather impractical to exhaustively test all possible crash states, which could be exponentially large.

To enable systematic testing of the large space of crash states, we use Long Short Term Memory (LSTM) [31], a machine learning model capable of learning long-term dependencies, to predict the file-system behavior on a crash state. We train the learning model to predict the probability that a file-system recovers to a consistent state after a crash.

The approach proposed here is not just confined to tackling the state-space explosion due to crashes in the middle of file-system operations. We could extend this idea to selectively test workloads of longer sequence lengths.

Learning Model. We use the simplest, sequential model which is a linear stack of layers. We build a three-layer network with the first one being an embedding layer with a maximum of 100 features, followed by an LSTM layer with 128 units and finally a single dense layer with sigmoid activation. The input to the learning model is a crash state, while a label representing if the file system recovers to a consistent state or not, is the output. Each crash state is represented as a sequence of IO requests, along with the information about the sector to which the IO request is issued, flags denoting whether it is a data or metadata write, the size of the write, and its data. When trained on this input data, the model learns the patterns in the sequence of IO requests in the crash state for a given file system, and thereby predicts the probability of it recovering to a consistent state. We use the binary cross-entropy loss function and Adam's optimizer [34] with learning rate ($r = 0.0001$) for our model. We train the model on a sample of 80% of the total data with a validation split of 0.20, for 20 epochs. Based on the probability predicted by the trained model, we determine if the crash state would be tested with CrashMonkey or not.

Using this approach, we filter out the crash states that could potentially not lead to crash-consistency bugs. Since each crash state takes about 1.5 seconds to be tested using CrashMonkey, filtering out obviously correct crash states is orders of magnitude faster than testing all of them with CrashMonkey. This approach thereby makes systematic testing of a exponentially large space of crash states feasible.

Table 7. Dataset Used by the LSTM Model

Type	Number of samples
Training set	6,678,024
Positive samples	27,185
Negative samples	6,650,839
Test set	1,781,199
Positive samples	480,576
Negative samples	1,300,623
Total samples	8,459,223

The table shows the breakdown of the dataset into training and test samples with the number of positive and negative samples in each set.

Table 8. LSTM Test Results

Measure	Percentage
Accuracy	88.253
False negative rate (FNR)	0.072
True positive rate (TPR)	99.928
True negative rate (TNR)	83.94
False positive rate (FPR)	16.06

The table shows the performance of the LSTM model in screening crash states to be tested with CrashMonkey.

Data Generation. We run CrashMonkey to collect around 40K crash states from 13 known bug-causing workloads in the btrfs file system. This is not a large enough dataset to train the model. Hence, we synthetically generate crash states in the following way. For every crash state, we permute the IO requests in the last epoch and assign the same label as that of the crash state to all its permutations. This is based on the assumption that, if a subset of IO requests resulted in a bug, then any permutation of the subset must also result in the bug, because the cause of the bug is the interaction between the IO requests in the subsets. This approach resulted in a total of 8.4 million crash states.

Results. We train the model for btrfs file system using the dataset of crash states from 13 known bugs. We uniformly sample the training and test data. The training set has roughly 80% of the data and the rest is used for testing. 20% of the training set is used for validation. To verify that the model generalizes, we ensure that the test set has buggy crash states which are not previously seen during training. A more detailed split-up of the dataset into training and test data, along with the number of positive and negative samples is presented in Table 7. Positive samples are the crash states that result in a bug, while negative samples are crash states that pass clean when tested with CrashMonkey. We train the model for 20 epochs using a threshold of 0.000005 (if the model predicts a probability above the threshold, then the corresponding crash state is tested with CrashMonkey). The model achieves an accuracy of 88% on the test data, with a very low false negative rate (FNR) of 0.072% as shown in Table 8. This is the lowest achievable FNR by the model. The low FNR signifies that we would miss a potentially bug-triggering crash state with a very low probability. The false positive rate (FPR) indicates that we would test 16% more crash states than the actual bug-triggering ones. If we reduce the FPR to 8%, the FNR shoots up to 23%. For the use

Table 9. File-System Persistence Guarantees

Persistence operation	Guarantees provided
ext4	
fsync (file)	Persists only a newly created file, not necessarily the hard links
fsync (directory)	Persists changes to the file names under the directory, but not file data
btrfs	
fsync (file)	Persists the file and all its hard links
fsync (directory)	Persists all changes to files and sub-directories under the directory

The table shows the difference in guarantees provided by file systems under different persistence scenarios. The btrfs file system expects to provide stronger guarantees than ext4.

case under consideration, it is important to not miss bugs; therefore, the low FNR is promising, even if it results in higher FPR.

8 DISCUSSION

The testing strategy employed by CrashMonkey to find new bugs was a result of our interaction with lead developers from mature Linux file systems like ext4, xfs, btrfs, and F2FS. We learned several useful lessons on what is deemed important by the developers and how to carefully make assumptions about the workload. This section is dedicated to sharing these lessons.

Developers Care about Bugs Violating Persistence Guarantees. Developers are motivated to resolve bugs that violate documented behavior. Most file systems today provide guarantees more than what the POSIX expects; however, these are not formally documented [85]. For example, in btrfs, `fsync()` of any file should be enough to persist that file in its current directory [63]; it does not require that its parent directory be explicitly persisted. Our conversations prompted developers to write down such guarantees explicitly for the first time [87]. Table 9 lists the guarantees for ext4 and btrfs file systems. Developers are motivated to resolve bugs violating these guarantees.

Not All File-System Operations Provide Well-Defined Crash-Consistency Guarantees. The set of file-system operations supported by Ace was refined based on our interaction with file-system developers. For example, the crash-consistency guarantees of symlink does not ensure that the symlink-ed file survives a crash, even if it was persisted explicitly before the crash [21]. This is because, unlike hard links, symlinks are not regular files and it is not possible to directly open them to `fsync()`. Understanding such semantics helped us generate workloads in a more reasonable manner, eliminating file-system operations like symlinks which provide no strong guarantees.

The Workload Triggering the Bug Should Be Realistic. Developers often question if the bug-triggering workloads would be a common case occurrence. If we report a bug that arises due to a specific interaction between a sequence of 100 file-system operations, it is important to back this workload by an application that would result in it. Or, the workload should have a severe consequence such as leaving the file system in an un-mountable state. Even when the consequences are severe, the developers expect that the initial file-system image be realistic. For example, the reports by syzbot [91], an automated Linux kernel fuzzer, are often criticized by the file-system developers [20, 22]. Unless the developers can come up with a regression test to reproduce the corrupt file-system image claimed by syzbot, they cannot identify the reason for corruption, nor provide a patch. Patching bugs of this kind are a low priority for the developers [61].

9 RELATED WORK

B^3 offers a new point in the spectrum of techniques addressing file-system crash consistency, alongside verified file systems and model checking. We now place B^3 in the context of prior approaches.

Verified File Systems. Recent work focuses on creating new, verified file systems from a specification [12, 13, 82]. These file systems are proven to have strong crash-consistency guarantees. However, the techniques employed are not useful for testing the crash consistency of existing, widely used Linux file systems written in low-level languages like C. The B^3 approach targets such file systems, which are not amenable to verification. Additionally, verified file systems can have bugs in either their specification or the unverified code. Testing approaches like B^3 are complimentary to verification techniques and can find bugs in verified file systems as well. CrashMonkey and Ace demonstrate this by finding a data loss bug in FSCQ.

Formal Crash-Consistency Models. Ferrite [9] formalizes crash-consistency models and can be used to test if a given ordering relationship holds in a file system; however, it is hard to determine what relationships to test. The authors used Ferrite to test a few simple relationships such as prefix append. On the other hand, Ace and CrashMonkey explore a wider range of workloads, and use oracles and developer-provided guarantees to automatically test correctness after a crash.

Model Checking. B^3 is closely related to in-situ model checking approaches such as EXPLODE [94] and FiSC [95]. However, unlike B^3, EXPLODE and FiSC require modifications to the buffer cache (to see all orderings of IO requests) and changes to the file-system code to expose choice points for efficient checking, a complex and time-consuming task. B^3 does not require changing any file-system code and it is conceptually simpler than in-situ model checking approaches, while still being effective at finding crash-consistency bugs.

Though the B^3 approach does not have the guarantees of verification or the power of model checking, it has the advantage of being easy to use (due to its black-box nature), being able to systematically test file systems (due to its exhaustive nature), and being able to catch crash-consistency bugs occurring on mature file systems.

Fuzzing. The B^3 approach bears some similarity to fuzz-testing techniques which explore inputs that will reveal bugs in the target system. The effectiveness of fuzzers is determined by the careful selection of uncommon inputs that would trigger exceptional behavior. However, B^3 does not randomize input selection. Neither does it use any sophisticated strategy to select workloads to test. Instead, B^3 exhaustively generates workloads in a bounded space, with the bounds informed by our study or provided by the user. While there exists fuzzers to test the correctness of system calls [24, 33, 74], there seem to be no fuzzing techniques to expose crash-consistency bugs. The effort by Nossum and Casasnovas [74] is closest to our work, where they generate file-system images that are likely to expose bugs during the normal operation of the file system (non-crash-consistency bugs).

Record-and-Replay Frameworks. CrashMonkey is similar to prior record-and-replay frameworks such as `dm-log-writes` [4], Block Order Breaker [76], and work by Zheng et al. [101]. Unlike `dm-log-writes`, which requires manual correctness tests or running `fsck`, CrashMonkey is able to automatically test crash consistency in an efficient manner.

Similar to CrashMonkey, the Block Order Breaker (BOB) [76] also creates crash states from recorded IO. However, BOB is only used to show that different file systems persist file-system operations in significantly different ways. The Application-Level Intelligent Crash Explorer (ALICE) explores application-level crash vulnerabilities in databases, key value stores, and so on. The major drawback with ALICE and BOB is that they require the user to handcraft workloads and

provide an appropriate checker for each workload. They lack systematic exploration of the workload space and do not understand persistence points, making it is extremely hard for a user to write bug-triggering workloads manually.

The logging and replay framework from Zheng et al. [101] is focused on testing whether databases provide ACID guarantees, works only on iSCSI disks, and uses only four workloads. CrashMonkey is able to test millions of workloads, and Ace allows us to generate a much wider range of workloads to test.

We previewed the ideas behind CrashMonkey in a workshop paper [62]. Since then, several features have been added to CrashMonkey with the prominent one being automatic crash-consistency testing. This article is an extended version of the OSDI'18 paper by Mohan et al. [73].

10 CONCLUSION

This article presents Bounded Black-Box Crash Testing (B^3), a new approach to testing file-system crash consistency. We study 26 crash-consistency bugs reported in Linux file systems over the past 5 years and find that most reported bugs could be exposed by testing small workloads in a systematic fashion. We exploit this insight to build two tools, CrashMonkey and Ace, that systematically test crash consistency. Running for 2 days on a research cluster of 65 machines, CrashMonkey and Ace reproduced 24 known bugs and found 10 new bugs in widely used Linux file systems, and a data loss bug in a verified file system.

We have made CrashMonkey and Ace available (with demo, documentation, and a single line command to run `seq-1` workloads) at https://github.com/utsaslab/crashmonkey. We encourage developers and researchers to test their file systems against the workloads included in the repository.

A APPENDIX

A.1 Bugs Reproduced by CrashMonkey and Ace

26 of the 28 known bugs are reproducible by CrashMonkey and Ace. We present the workload that triggers each bug, along with the file system on which it occurs, the difference in expected and actual states after a crash, and the overall consequence of the bug.

Workload 1	Details [54]	
mkdir A write(0-16K) A/foo sync mv A/foo A/bar write (0-4K) A/foo fsync A/foo —Crash—	**File system**	btrfs F2FS
	Expected	A/foo : Size 4K A/bar : Size 16K
	Actual	A/foo : Size 4K
	Consequence	Persisted file missing

Workload 2	Details [27]	
write (0-8K) foo fsync foo falloc -k (8-16K) foo fdatasync foo —Crash—	**File system**	ext4, F2FS
	Expected	foo: 32 sectors
	Actual	foo: 16 sectors
	Consequence	Blocks allocated beyond EOF are lost

Workload 3	Details [56]	
mkdir A mkfifo A/foo touch A/baz fsync A/baz mv A/foo A/bar link A/bar A/foo remove A/baz fsync A/baz —Crash—	**File system**	btrfs
	Expected	A/foo A/bar
	Actual	FS unmountable
	Consequence	FS unmountable

Workload 4	Details [28]	
write (16-20K) foo d-write (0-4K) foo —Crash—	**File system**	ext4
	Expected	foo: Size 4K
	Actual	foo: Size 0
	Consequence	Metadata inconsistent

Workload 5	Details [60]	
mkdir A touch A/foo link A/foo A/bar sync unlink A/bar touch A/bar fsync A/bar —Crash—	**File system**	btrfs
	Expected	A/foo A/bar
	Actual	FS unmountable
	Consequence	FS unmountable

Workload 6	Details [8]	
mkdir A touch A/foo fsync A/foo —Crash—	**File system**	btrfs
	Expected	Writable FS
	Actual	Cannot create files
	Consequence	Cannot create new files

Workload 7	Details [49]	
mkdir A,B,C touch A/foo link A/foo B/foo1 touch B/bar sync unlink B/foo1 mv B/bar C/bar fsync A/foo —Crash—	**File system**	btrfs
	Expected	C/bar or B/bar
	Actual	C/bar and B/bar missing
	Consequence	Persisted file missing

Workload 8	Details [53]	
mkdir -p A/B mkdir A/C touch A/B/foo touch A/B/bar sync mv A/B A/C mkdir A/B fsync A/B —Crash—	**File system**	btrfs
	Expected	A/B A/C/foo A/C/bar
	Actual	A/B
	Consequence	Directory and its contents missing

Workload 9	Details [50]	
mkdir A,B touch A/foo mkdir B/C touch B/baz sync link A/foo A/bar mv B/C A/ mv B/baz A/ fsync A/foo —Crash—	**File system**	btrfs
	Expected	A/C or B/C A/baz or B/baz
	Actual	A/C & B/C A/baz & B/baz
	Consequence	File persisted in both directories

Workload 10	Details [29]	
mkdir A sync symlink foo, A/bar fsync A —Crash—	**File system**	btrfs
	Expected	A/bar must point to foo
	Actual	A/bar is empty
	Consequence	Empty symlink

Workload 11	Details [52]	
mkdir A touch A/foo fsync A fsync A/foo mv A/foo A/bar touch A/foo fsync A/bar —Crash—	**File system**	btrfs
	Expected	A/foo A/bar
	Actual	A/bar
	Consequence	Persisted file missing

Workload 12	Details [45]	
write(0-132K) foo punch_hole(96-128K) punch_hole(64-192K) punch_hole(32-128K) fsync foo —Crash—	**File system**	btrfs
	Expected	Hole: 32–192K
	Actual	Hole: 32–128K
	Consequence	Extent map incorrect

Workload 13	Details [47]	
mkdir A touch A/foo touch A/bar sync link A/foo A/foo1 link A/bar A/bar1 fsync A/bar —Crash—	**File system**	btrfs
	Expected	Writable FS
	Actual	Dir A unremovable
	Consequence	Directory unremovable

Workload 14	Details [40]	
touch foo write (0-256K) foo sync mmap (0-256K) foo m-write (0-4K) m-write (252-256K) msync (0-64K) msync(192-256K) —Crash—	**File system**	btrfs
	Expected	Both writes persist
	Actual	Second write not persisted
	Consequence	Data loss

Workload 15	Details [46]	
mkdir A sync touch A/foo link A/foo A/bar sync remove A/bar fsync A/foo —Crash—	**File system**	btrfs
	Expected	Writable FS
	Actual	Dir A un-removable
	Consequence	Directory unremovable

Workload 16	Details [43]	
mkdir A touch A/foo sync write (0-16K) A/foo fsync A/foo link A/foo A/bar —Crash—	**File system**	btrfs
	Expected	foo: Size 16K
	Actual	foo: Size 0
	Consequence	Data loss

Workload 17	Details [42]	
write(0-16K) foo fsync foo sync punch_hole -k (8,000–12,096) foo fsync foo —Crash—	**File system**	btrfs
	Expected	Hole must persist
	Actual	Hole not persisted
	Consequence	Punch_hole does not persist

Workload 18	Details [48]	
touch foo setxattr foo u1 val1 setxattr foo u2 val2 setxattr foo u3 val3 sync removexattr foo u2 fsync foo —Crash—	**File system**	btrfs
	Expected	u1 u2
	Actual	u1 u2 u3
	Consequence	Remove xattr does not persist

Workload 19	Details [26]	
mkdir A touch A/foo sync link A/foo A/bar1 link A/foo A/bar2 sync unlink A/bar2 fsync A/foo —Crash—	**File system**	btrfs
	Expected	Writable FS
	Actual	Dir A unremovable
	Consequence	Dir unremovable

Workload 20	Details [51]	
mkdir -p A/B, C touch A/B/foo sync mv A/B/foo C/foo touch A/bar fsync A —Crash—	**File system**	btrfs
	Expected	A/bar C/foo
	Actual	A/bar A/B/foo
	Consequence	Renamed file missing

Workload 21	Details [39]	
mkdir A touch A/foo sync touch A/bar fsync A fsync A/bar —Crash—	**File system**	btrfs
	Expected	Writable FS
	Actual	Dir A unremovable
	Consequence	Directory unremovable

Workload 22	Details [5]	
touch A/foo write (0-4K) A/foo sync mv A/foo A/bar fsync A/bar —Crash—	**File system**	btrfs
	Expected	A/bar
	Actual	A/foo
	Consequence	Persisted file missing

Workload 23	Details [44]	
write(0-32K) foo sync link foo, bar sync write(32-64K) foo fsync foo —Crash—	**File system**	btrfs
	Expected	foo: Size 64K
	Actual	foo: Size 32K
	Consequence	Data loss

Workload 24	Details [6]	
touch foo mkdir A fsync foo sync mv foo A/bar fsync A fsync A/bar —Crash—	**File system**	btrfs
	Expected	Writable FS
	Actual	Dir A unremovable
	Consequence	Directory unremovable

A.2 New Bugs Found by CrashMonkey and Ace

CrashMonkey and Ace found 10 new bugs across widely used Linux file systems, btrfs and F2FS. Additionally, our tools also found a data loss in FSCQ, ascertaining the fact that unverified components of a verified file system could lead to bugs in the final artifact.

Workload 1	Details [64]	
mkdir A touch A/bar fsync A/bar mkdir B touch B/bar rename B/bar A/bar touch A/foo fsync A/foo fsync A —Crash—	**File system**	btrfs
	Expected	A/foo A/bar or B/bar
	Actual	A/foo
	Consequence	Previously persisted file goes missing

Workload 2	Details [58]	
mkdir A mkdir A/C rename A/C B touch B/bar fsync B/bar rename B/bar A/bar rename A B fsync B/bar —Crash—	**File system**	btrfs
	Expected	B/bar
	Actual	A/bar B/bar
	Consequence	Rename persists the file in both directories

Workload 3	Details [59]	
mkdir A mkdir B mkdir A/C touch B/foo fsync B/foo link B/foo A/C/foo fsync A —Crash—	**File system**	btrfs
	Expected	B/foo A/C/foo
	Actual	B/foo
	Consequence	Persisted directory missing

Workload 4	Details [63]	
mkdir A sync rename A B touch B/foo fsync B/foo fsync B —Crash—	**File system**	btrfs
	Expected	B B/foo
	Actual	A/foo
	Consequence	Persisted file missing

Workload 5	Details [63]	
mkdir A mkdir B touch A/foo link (A/foo, B/foo) fsync A/foo fsync B/foo —Crash—	**File system**	btrfs
	Expected	A/foo B/foo
	Actual	A/foo
	Consequence	Hard link missing even after persisting both files involved

Workload 6	Details [15]	
mkdir test mkdir test/A touch test/foo touch test/A/foo fsync test/A/foo fsync test —Crash—	**File system**	btrfs
	Expected	test/A/foo test/foo
	Actual	test/A/foo
	Consequence	File missing in spite of persisting parent directory

Workload 7	Details [16]	
touch foo mkdir A link (foo, A/bar) fsync foo —Crash—	**File system**	btrfs
	Expected	foo A/bar
	Actual	foo
	Consequence	Fsync of a file does not persist all its names

Workload 8	Details [55]	
write (0–16K) foo fsync foo falloc -k (16–20K) fsync(foo) —Crash—	**File system**	btrfs
	Expected	foo: 40 sectors
	Actual	foo: 32 sectors
	Consequence	Blocks allocated beyond EOF are lost

Workload 9	Details [98]	
write (0–16K) foo fsync foo fzero -k (16–20K) fsync(foo) —Crash—	**File system**	F2FS
	Expected	foo: Size 16K
	Actual	foo: Size 20K
	Consequence	Recovers to incorrect file size

Workload 10	Details [97]	
mkdir A sync rename A B touch B/foo fsync B/foo —Crash—	**File system**	F2FS
	Expected	B/foo
	Actual	A/foo
	Consequence	Persisted file ends up in a different directory

Workload 11	Details [11]	
write (0–4K) foo sync write (4–8K) foo fdatasync foo —Crash—	**File system**	FSCQ
	Expected	foo: Size 8K
	Actual	foo: Size 4K
	Consequence	Data loss

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, and the members of Systems and Storage Lab and LASR group for their feedback and guidance. We would like to thank Sonika Garg, Subrat Mainali, and Fabio Domingues for their contributions to the CrashMonkey codebase. We are thankful to Amir Goldstein and Ted Ts'o who encouraged us in doing this work. We also thank the Chameleon Cloud team for providing a research cluster to test the workloads using CrashMonkey. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not reflect the views of other institutions.

REFERENCES

[1] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. 2017. Evolving ext4 for shingled disks. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 105–120. https://www.usenix.org/conference/fast17/technical-sessions/presentation/aghayev.

[2] Amazon. 2018. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[3] Apple. 2018. fsync(2) Mac OS X Developer Tools Manual Page. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/fsync.2.html.

[4] Josef Bacik. 2018. dm: Log Writes Target. https://www.redhat.com/archives/dm-devel/2014-December/msg00047.html.

[5] Josef Bacik. 2013. xfstests: Add a Rename fsync Test. https://patchwork.kernel.org/patch/3234541/.

[6] Josef Bacik. 2013. xfstests: Add Generic/321 to Test fsync() on Directories V2. https://patchwork.kernel.org/patch/3234531/.

[7] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 69–86. DOI: https://doi.org/10.1145/3132747.3132779

[8] Liu Bo. 2018. btrfs: Fix Unexpected -EEXIST When Creating New Inode. https://patchwork.kernel.org/patch/10184679/.
[9] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. 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'16)*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 83–98. DOI: https://doi.org/10.1145/2872362.2872406
[10] btrfs Wiki. 2018. btrfs Check. https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs-check.
[11] Tej Chajed. 2018. Disable Logged Writes for Crash Safety. https://github.com/mit-pdos/fscq/commit/97b50eceedf15a2c82ce1a5cf83c231eb3184760.
[12] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 270–286. DOI: https://doi.org/10.1145/3132747.3132776
[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Ethan L. Miller and Steven Hand (Eds.). ACM, 18–37. DOI: https://doi.org/10.1145/2815400.2815402
[14] Vijay Chidambaram. 2015. *Orderless and Eventually Durable File Systems*. Ph.D. Dissertation. University of Wisconsin, Madison.
[15] Vijay Chidambaram. 2018. btrfs: Strange Behavior (Possible Bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77929.html.
[16] Vijay Chidambaram. 2018. btrfs: Symlink Not Persisted Even After fsync. https://www.spinics.net/lists/fstests/msg09379.html.
[17] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.
[18] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*. 101–116.
[19] Dave Chinner. 2018. btrfs: Symlink Not Persisted Even After fsync. https://www.spinics.net/lists/fstests/msg09363.html.
[20] Dave Chinner. 2018. INFO: Task Hung in xlog_grant_head_check. https://lkml.org/lkml/2018/5/22/1056.
[21] Dave Chinner. 2018. Symlink Not Persisted Even After fsync. https://www.spinics.net/lists/linux-btrfs/msg76835.html.
[22] Dave Chinner. 2018. WARNING: Bad Unlock Balance in xfs_iunlock. https://lkml.org/lkml/2018/4/3/9.
[23] Jonathan Corbet. 2014. Toward Better Testing. https://lwn.net/Articles/591985/.
[24] David Drysdale. 2016. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News* 2 (2016), 33.
[25] The Open Group. 2018. The Open Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/.
[26] Eryu Guan. 2015. fstests: Generic Test for fsync of File with Multiple Links. https://www.spinics.net/lists/linux-btrfs/msg45915.html.
[27] Eryu Guan. 2017. ext4: fix fdatasync(2) After fallocate(2) Operation. https://lore.kernel.org/patchwork/patch/864704/.
[28] Eryu Guan. 2018. ext4: Update idisksize if Direct Write Past Ondisk Size. https://marc.info/?l=linux-ext4&m=151669669030547&w=2.
[29] Greg Kroah Hartman. 2016. btrfs: Fix Empty Symlink After Creating Symlink and fsync Parent dir. https://lore.kernel.org/patchwork/patch/685585/.
[30] Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS File server appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*.
[31] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[32] Yige Hu, Zhiting Zhu, Ian Neal, Yougjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. 2018. TxFS: Leveraging file-system crash consistency to provide ACID transactions. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*. USENIX Association.
[33] Dave Jones. 2011. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium*.
[34] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *Arxiv:1412.6980*.
[35] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High performance metadata integrity protection in the WAFL copy-on-write file system. In *Proceedings of the 15th USENIX Conference on File and Storage*

Technologies (FAST'17). USENIX Association, 197–212. https://www.usenix.org/conference/fast17/technical-sessions/presentation/kumar.

[36] Ubuntu Bugs LaunchPad. 2018. Bug #317781: Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.

[37] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 273–286. http://dl.acm.org/citation.cfm?id=2750482.2750503.

[38] Joe Mambretti, Jim Chen, and Fei Yeh. 2015. Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn). In *Proceedings of the 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI'15)*. IEEE, 73–79.

[39] Filipe Manana. 2014. btrfs: Fix Directory Recovery from fsyncLog. https://patchwork.kernel.org/patch/4864571/.

[40] Filipe Manana. 2014. btrfs: Fix fsync Data Loss after a Ranged fsync. https://patchwork.kernel.org/patch/4813651/.

[41] Filipe Manana. 2015. btrfs: Add Missing Inode Update When Punching Hole. https://patchwork.kernel.org/patch/5830801/.

[42] Filipe Manana. 2015. btrfs: Add Missing Inode Update When Punching Hole. https://www.spinics.net/lists/linux-btrfs/msg42047.html.

[43] Filipe Manana. 2015. btrfs: Fix fsync Data Loss After Adding Hard Link to Inode. https://patchwork.kernel.org/patch/5822681/.

[44] Filipe Manana. 2015. btrfs: Fix fsync Data Loss After Append Write. https://patchwork.kernel.org/patch/6624481/.

[45] Filipe Manana. 2015. btrfs: Fix Hole Punching When Using the No-Holes Feature. https://patchwork.kernel.org/patch/7536021/.

[46] Filipe Manana. 2015. btrfs: Fix Metadata Inconsistencies After Directory fsync. https://patchwork.kernel.org/patch/6058101/.

[47] Filipe Manana. 2015. btrfs: Fix Stale Directory Entries After fsync Log Replay. https://patchwork.kernel.org/patch/6852751/.

[48] Filipe Manana. 2015. btrfs: Remove Deleted xattrs on fsync Log Replay. https://www.spinics.net/lists/linux-btrfs/msg42162.html.

[49] Filipe Manana. 2016. btrfs: Fix File Loss on Log Replay After Renaming a File and fsync. https://patchwork.kernel.org/patch/8293181/.

[50] Filipe Manana. 2016. btrfs: Fix for Incorrect Directory Entries After fsync Log Replay. https://patchwork.kernel.org/patch/8766401/.

[51] Filipe Manana. 2016. fstests: Generic Test for Directory fsync After Rename Operation. https://patchwork.kernel.org/patch/8312681/.

[52] Filipe Manana. 2016. fstests: Generic Test for fsync After File Rename. https://patchwork.kernel.org/patch/9297215/.

[53] Filipe Manana. 2016. fstests: Generic Test for fsync After Renaming Directory. https://patchwork.kernel.org/patch/8694291/.

[54] Filipe Manana. 2016. fstests: Generic Test for fsync After Renaming File. https://patchwork.kernel.org/patch/8694301/.

[55] Filipe Manana. 2018. btrfs: Blocks Allocated Beyond EOF are Lost. https://www.spinics.net/lists/linux-btrfs/msg75108.html.

[56] Filipe Manana. 2018. Btrfs: Fix Log Replay Failure After Linking Special File and fsync. https://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg73890.html.

[57] Filipe Manana. 2018. btrfs: Fix Log Replay Failure after Unlink and Link Combination. https://www.spinics.net/lists/linux-btrfs/msg75204.html.

[58] Filipe Manana. 2018. btrfs: Strange Behavior (Possible Bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg81425.html.

[59] Filipe Manana. 2018. btrfs: Sync Log After Logging New Name. https://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg77875.html.

[60] Filipe Manana. 2018. Generic: Test fsync New File After Removing Hard Link. https://marc.info/?l=linux-btrfs&m=151983349112005&w=2.

[61] Filipe Manana. 2018. Re: Strange Behavior (Possible Bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg81425.html.

[62] Ashlie Martinez and Vijay Chidambaram. 2017. Crashmonkey: A framework to systematically test file-system crash consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*. USENIX Association, 6–6.

[63] Chris Mason. 2018. btrfs: Inconsistent Behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77282.html.

[64] Chris Mason. 2018. btrfs: Inconsistent Behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77318.html.
[65] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, Vol. 2. 21–33.
[66] Marshall K. McKusick, Gregory R. Ganger, et al. 1999. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 1–17.
[67] R. McMillan. 2012. Amazon Blames Generators for Blackout that Crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazon explains/.
[68] R. Miller. 2012. Power Outage Hits London Data Center. http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/.
[69] R. Miller. 2013. Data Center Outage Cited in Visa Downtime Across Canada. http://www.datacenterknowledge.com/archives/2013/01/28/data-center-outage-cited-in-visa-downtime-across-canada/.
[70] R. Miller. 2013. Power Outage Knocks Dreamhost Customers Offline. http://www.datacenterknowledge.com/archives/2013/03/20/power-outage- knocks-dreamhost-customers-offline/.
[71] Jayashree Mohan. 2018. btrfs: Hard Link Not Persisted On fsync. https://www.spinics.net/lists/linux-btrfs/msg76878.html.
[72] Jayashree Mohan. 2018. btrfs: Inconsistent Behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77219.html.
[73] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, 33–50. https://www.usenix.org/conference/osdi18/presentation/mohan.
[74] Vegard Nossum and Quentin Casasnovas. 2016. Filesystem Fuzzing with American Fuzzy Lop. https://lwn.net/Articles/685182/.
[75] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 181–196. https://www.usenix.org/conference/fast17/technical-sessions/presentation/pillai.
[76] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.
[77] POSIX. 2018. fsync: The Open Group Base Specifications Issue 6. http://pubs.opengroup.org/onlinepubs/009695399/functions/fsync.html.
[78] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*. 105–120.
[79] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2018. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 257–264. https://www.usenix.org/conference/fast18/presentation/rho.
[80] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 9.
[81] Mendel Rosenblum and John Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (Feb. 1992), 26–52.
[82] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 1–16. http://dl.acm.org/citation.cfm?id=3026877.3026879.
[83] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. 2018. High-performance transaction processing in journaling file systems. In *Proceedings of the16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 227–240. https://www.usenix.org/conference/fast18/presentation/son.
[84] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference (USENIX'96)*.
[85] Ted Ts'o. 2018. Symlink Not Persisted Even After fsync. https://www.spinics.net/lists/linux-btrfs/msg76844.html.
[86] Theodore Y. Ts'o. 2018. btrfs: Inconsistent Behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77389.html.
[87] Theodore Y. Ts'o. 2018. btrfs: Inconsistent Behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77340.html.

[88] UTSASLab. 2018. Crash-Consistency Bugs Studied and Reproduced. https://github.com/utsaslab/crashmonkey/blob/master/reproducedBugs.md.
[89] UTSASLab. 2018. New Crash-Consistency Bugs Found. https://github.com/utsaslab/crashmonkey/blob/master/newBugs.md.
[90] J. Verge. 2014. Internap Data Center Outage Takes Down Livestream and Stackexchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/.
[91] Dmitry Vyukov. 2018. Syzbot. https://lwn.net/Articles/749910/.
[92] R. S. V. Wolffradt. 2014. Fire In Your Data Center: No Power, No Access, Now What? http://www.govtech.com/state/Fire-in-your-Data-Center-No-Power-No-Access-Now-What.html.
[93] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 211–226. http://dl.acm.org/citation.cfm?id=3189759.3189779.
[94] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*.
[95] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. 2004. Using model checking to find serious file system errors (awarded best paper!). In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 273–288. http://www.usenix.org/events/osdi04/tech/yang.html.
[96] Chao Yu. 2017. f2fs: Keep isize Once Block is Reserved Cross EOF. https://sourceforge.net/p/linux-f2fs/mailman/message/36104201/.
[97] Chao Yu. 2018. f2fs: Enforce fsync_mode=strict for Renamed Directory. https://lkml.org/lkml/2018/4/25/674.
[98] Chao Yu. 2018. f2fs: Fix to Set KEEP_SIZE Bit in f2fs_zero_range. https://lore.kernel.org/patchwork/patch/889955/.
[99] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 1–14. https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan.
[100] Shuanglong Zhang, Helen Catanese, and Andy An-I Wang. 2016. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 15–22. https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-shuanglong.
[101] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 449–464. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai.

Received November 2018; accepted March 2019