Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing

Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, and Pandian Raju, ***University of Texas at Austin;***
Vijay Chidambaram, ***University of Texas at Austin and VMware Research***

https://www.usenix.org/conference/osdi18/presentation/mohan

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing

Jayashree Mohan*[1] Ashlie Martinez*[1] Soujanya Ponnapalli[1] Pandian Raju[1]
Vijay Chidambaram[1,2]

[1]University of Texas at Austin [2]VMware Research

Abstract

We present a new approach to testing 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. Each workload is tested 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. B^3 builds upon insights derived from our study of crash-consistency bugs reported in Linux file systems in the last five 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. We build two tools, CRASHMONKEY and ACE, to demonstrate the effectiveness of this approach. Our tools are able to find 24 out of the 26 crash-consistency bugs reported in the last five years. Our tools also revealed 10 *new* crash-consistency bugs in widely-used, mature Linux file systems, seven of which existed in the kernel since 2014. The new bugs result in severe consequences like broken rename atomicity and loss of persisted files.

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 [37] in 2009, they introduced a crash-consistency bug that led to wide-spread data loss [24]. Given the potential consequences of crash-consistency bugs and the fact that even professionally-managed datacenters occasionally suffer from power losses [39–42, 60, 61], 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 [55], btrfs [51], and F2FS [25]. 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` [16], 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 paper 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 five 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()`. We have reported all new bugs; developers have submitted patches for four, 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 five years (§3). Our study provided key insights

*Both authors contributed equally

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 explore 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.

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 it's 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 [8, 9, 53] and model checking [63, 64]. 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 [62].

This paper 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 five years (§3)
- The bounded black-box crash testing approach (§4)
- The design and implementation of CrashMonkey and Ace[1] (§5)
- Experimental results demonstrating that our tools are able to efficiently find existing and new bugs across widely-used Linux file systems (§6)

2 Background

We first provide some background on file-system crash consistency, why crash-consistency bugs occur, and why it is important to test file-system crash consistency.

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 [10, 38].

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

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 shouldn't 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 paper, 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 [15, 56]. 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 [51]. 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 [28]. 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 [24]. 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.

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

Figure 1: **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 [33]. The bug causes the file system to become un-mountable.

POSIX and file-system guarantees. Nominally, Linux file systems implement the POSIX API, providing guarantees as laid out in the POSIX standard [18]. Unfortunately, POSIX is extremely vague. For example, under POSIX it is legal for `fsync()` to *not* make data durable [48]. 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 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 1 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 [7]. 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 [13, 14, 46] and designing new file systems that increase performance [1, 5, 21, 23, 50, 54, 68, 69]. 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` [16] 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

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

Kernel Version	# bugs
3.12	3
3.13	9
3.16	1
4.1.1	2
4.4	9
4.15	3
4.16 (latest)	1
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

Table 1: **Analyzing crash-consistency bugs**. The tables break down the 26 unique crash-consistency bugs reported over the last five years (since 2013) by different criteria. Two bugs were reported on two different file systems, leading to a total of 28 bugs.

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 five years on widely-used Linux file systems [58]. 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.

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 [27] 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 [29] 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 [19]) 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 *seven* years after it was introduced. The bug was caused by incorrectly processing a hard link in

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)`

Table 2: **Examples of crash-consistency bugs**. The table shows some of the crash-consistency bugs reported in the last five years. The bugs have severe consequences, ranging from losing user data to making directories un-removable.

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 [30]. 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 *four 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 [31]. 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 specific setup: 3000 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 is 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 is 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 file, *etc.* 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 [49] or copy-on-write [20, 52], 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.
3. B^3 focuses on workloads where files and directories are explicitly persisted. If we created a file, waited one 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.

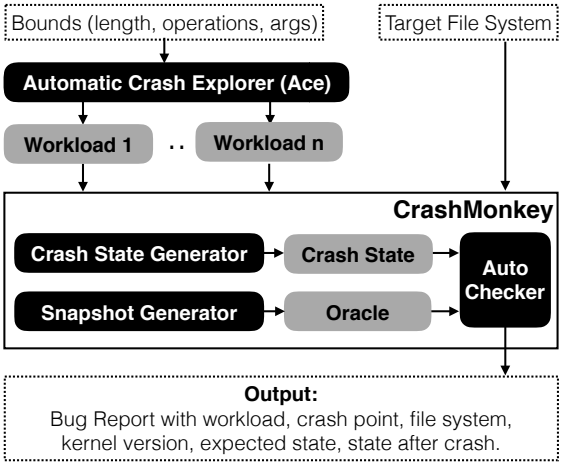

Figure 2: **System architecture**. Given bounds for exploration, ACE generates a set of workloads. Each workload is then fed to CRASHMONKEY, which generates a set of crash states and corresponding oracles. The AutoChecker compares persisted files in each oracle/crash state pair; a mismatch indicates a bug.

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 2, 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.

Overview. CRASHMONKEY operates in three phases as shown in Figure 3. In the first phase, CRASHMONKEY profiles the workload by collecting information about all file-system operations and IO requests made during the

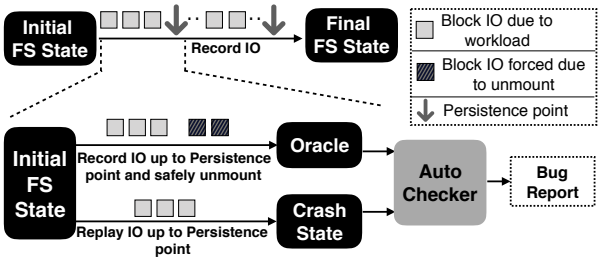

Figure 3: **CRASHMONKEY operation**. CRASHMONKEY first records the block IO requests that the workload translates to, capturing reference images called oracles after each persistence point. CRASHMONKEY then generates crash states by replaying the recorded IO and tests for consistency against the corresponding oracle.

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. 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 1300 lines of C code which can be compiled and inserted into the kernel at run time, thus avoiding the need for long kernel re-compilations. The user-space utilities consist of 4800 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 §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

B^3 bound	Insight from the study	Bound chosen by ACE
Number of operations	Small workloads of 2-3 core operations	Maximum # of core ops in a workload is *three*
Files and directories	Reuse file and directory names	2 directories of depth 2, each with 2 unique files
Data operations	Coarse grained, overlapping ranges of writes	Overwrites to start, middle & 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

Table 3: **Bounds used by ACE**. The table shows the specific values picked by ACE for each B^3 bound.

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 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 unmountable. 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 [27].

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 [57]. 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 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 4.

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 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 4 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 4 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 2500 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 direc-

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

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

Figure 4: **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.

tories 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 increase 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.

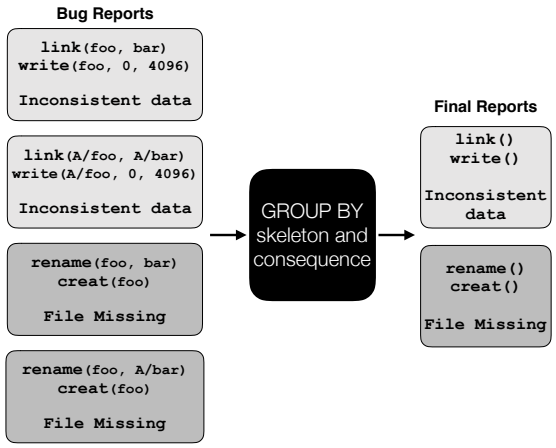

Figure 5: **Post-processing**. The figure shows how generated bug reports are processed to eliminate duplicates.

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 5. Using the skeleton instead of the fully fleshed-out workload allows 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? (§6.2)
- What is the performance of CrashMonkey? (§6.3)
- What is the performance of Ace? (§6.4)
- How much memory and CPU does CrashMonkey consume? (§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 [26], an experimental testbed for large-scale computation.

We employ a cluster of 65 nodes on Chameleon Cloud. Each node has 40 cores, 48 GB RAM, and 128 GB SSD. We install 12 VirtualBox virtual machines running Ubuntu 16.04 LTS on each node, each with 2 GB RAM and 10 GB storage. Each virtual machine runs one instance of CrashMonkey. Thus, we have a total of 780 virtual machines testing workloads with CrashMonkey in parallel. We found we are limited to 780 virtual machines by the storage available to each physical node.

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.

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 focused on F2FS and btrfs for the larger `seq-3` workloads. In total, we spend 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.

Results. CrashMonkey and Ace found 10 **new** crash-consistency bugs [59] in btrfs and F2FS, in addition to reproducing 24 out of 26 bugs reported over the past five 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 [11, 12, 43, 44]. Developers have

Sequence type	File-system operations tested	# of workloads	Run time (minutes)
seq-1	creat, mkdir, falloc, buffered write, mmap, link, direct-IO write, unlink, rmdir, setxattr, removexattr, remove, unlink, truncate	300	1
seq-2		254K	215
seq-3-data	buffered write, mmap, direct-IO write, falloc	120K	102
seq-3-metadata	buffered write, link, unlink, rename	1.5M	1274
seq-3-nested	link, rename	1.5M	1274
Total		3.37M	2866

Table 4: **Workloads tested**. 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 two days, reproducing 24 known bugs and finding 10 new crash-consistency bugs.

submitted patches for four bugs [32, 35, 66, 67], and are working on patches for the others [34]. 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 [65]; however, the `fallocate()` system call has several more flags like `ZERO_RANGE`, `PUNCH_HOLE`, *etc.*, 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 two 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 don't always involve obvious sequences of operations.

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

Table 5: **Newly discovered bugs**. The table shows the new bugs found by CRASHMONKEY and ACE. The bugs have severe consequences, ranging from losing allocated blocks 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 *.

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 two 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 20 ms to construct each crash state. Furthermore, since CRASHMONKEY uses fine-grained correctness tests, checking crash consistency with both read and write tests takes only 20 ms.

6.4 Ace Performance

ACE generated all the workloads that were tested (3.37M) in 374 minutes ($\approx$ 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 is 20.12 MB. 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 (480 KB per workload). Finally, the CPU consumption of CRASHMONKEY, as reported by `top`, was negligible (less than 1 percent).

7 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 [8, 9, 53]. 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.

Formal Crash-Consistency Models. Ferrite [6] 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 [63] and FiSC [64]. 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 [17, 22, 45], there seem to be no fuzzing techniques to expose crash-consistency bugs. The effort by Nossum and Casasnovas [45] 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 [47], and work by Zheng *et al.* [70]. 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) [47] 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 *etc.* 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.* [70] 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 ranger of workloads to test.

We previewed the ideas behind CRASHMONKEY in a workshop paper [36]. Since then, several features have been added to CRASHMONKEY with the prominent one being automatic crash-consistency testing.

8 Conclusion

This paper 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 five 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 two 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.

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.

Acknowledgments

We would like to thank our shepherd, Angela Demke Brown, 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. This work was supported by generous donations from VMware, Google, and Facebook. 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] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers. Evolving ext4 for shingled disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 105–120, Santa Clara, CA, 2017. USENIX Association.

[2] Amazon. Amazon ec2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[3] Apple. fsync(2) mac os x developer tools manual page. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/fsync.2.html.

[4] J. Bacik. dm: log writes target. https://www.redhat.com/archives/dm-devel/2014-December/msg00047.html.

[5] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 69–86. ACM, 2017.

[6] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In T. Conte and Y. Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 83–98. ACM, 2016.

[7] btrfs Wiki. btrfs check. https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs-check.

[8] H. Chen, T. Chajed, A. Konradi, S. Wang, A. Ileri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 270–286. ACM, 2017.

[9] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In E. L. Miller and S. Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 18–37. ACM, 2015.

[10] V. Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2015.

[11] V. Chidambaram. btrfs: strange behavior (possible bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77929.html, Apr 2018.

[12] V. Chidambaram. btrfs: symlink not persisted even after fsync. https://www.spinics.net/lists/fstests/msg09379.html, Apr 2018.

[13] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[14] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, Feb. 2012.

[15] D. Chinner. btrfs: symlink not persisted even after fsync. https://www.spinics.net/lists/fstests/msg09363.html, Apr 2018.

[16] J. Corbet. Toward better testing. https://lwn.net/Articles/591985/, 2014.

[17] D. Drysdale. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News*, 2:33, 2016.

[18] T. O. Group. The open group base specifications issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[19] E. Guan. ext4: update idisksize if direct write past ondisk size. https://marc.info/?l=linux-ext4&m=151669669030547&w=2, Jan 2018.

[20] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.

[21] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *The 2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, 2018. USENIX Association.

[22] D. Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium, pages*, 2011.

[23] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[24] U. B. LaunchPad. Bug #317781: Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.

[25] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[26] J. Mambretti, J. Chen, and F. Yeh. Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn). In *Cloud Computing Research and Innovation (ICCCRI), 2015 International Conference on*, pages 73–79. IEEE, 2015.

[27] F. Manana. btrfs: fix directory recovery from fsync log. https://patchwork.kernel.org/patch/4864571/, Sep 2014.

[28] F. Manana. btrfs: add missing inode update when punching hole. https://patchwork.kernel.org/patch/5830801/, Feb 2015.

[29] F. Manana. btrfs: fix fsync data loss after adding hard link to inode. https://patchwork.kernel.org/patch/5822681/, Feb 2015.

[30] F. Manana. btrfs: fix metadata inconsistencies after directory fsync. https://patchwork.kernel.org/patch/6058101/, March 2015.

[31] F. Manana. btrfs: fix stale directory entries after fsync log replay. https://patchwork.kernel.org/patch/6852751/, July 2015.

[32] F. Manana. btrfs: blocks allocated beyond eof are lost. https://www.spinics.net/lists/linux-btrfs/msg75108.html, Feb 2018.

[33] F. Manana. btrfs: fix log replay failure after unlink and link combination. https://www.spinics.net/lists/linux-btrfs/msg75204.html, Feb 2018.

[34] F. Manana. btrfs: strange behavior (possible bugs) in btrfs. https://www.spinics.net/lists/linux-btrfs/msg81425.html, Aug 2018.

[35] F. Manana. btrfs: sync log after logging new name. https://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg77875.html, Jun 2018.

[36] A. Martinez and V. Chidambaram. 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*, pages 6–6. USENIX Association, 2017.

[37] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[38] M. K. McKusick, G. R. Ganger, et al. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, 1999.

[39] R. McMillan. Amazon Blames Generators For Blackout That Crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazonexplains/, 2012.

[40] R. Miller. Power Outage Hits London Data Center. http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/, 2012.

[41] R. Miller. 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/, 2013.

[42] R. Miller. Power Outage Knocks Dreamhost Customers Offline. http://www.datacenterknowledge.com/archives/2013/03/20/power-outage-knocks-dreamhost-customers-offline/, 2013.

[43] J. Mohan. btrfs: hard link not persisted on fsync. https://www.spinics.net/lists/linux-btrfs/msg76878.html, Apr 2018.

[44] J. Mohan. btrfs: inconsistent behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77219.html, Apr 2018.

[45] V. Nossum and Q. Casasnovas. Filesystem fuzzing with american fuzzy lop. https://lwn.net/Articles/685182/, 2016.

[46] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 181–196, Santa Clara, CA, 2017. USENIX Association.

[47] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 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)*, Broomfield, CO, October 2014.

[48] POSIX. fsync: The open group base specifications issue 6. http://pubs.opengroup.org/onlinepubs/009695399/functions/fsync.html.

[49] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[50] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J. Hwang, S. Cho, D. D. Lee, and J. Jeong. Fstream: Managing flash streams in the file system. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 257–264, Oakland, CA, 2018. USENIX Association.

[51] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[52] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[53] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 1–16, Berkeley, CA, USA, 2016. USENIX Association.

[54] Y. Son, S. Kim, H. Y. Yeom, and H. Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240, Oakland, CA, 2018. USENIX Association.

[55] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, Jan. 1996.

[56] T. Y. Ts'o. btrfs: Inconsistent behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77389.html, Apr 2018.

[57] T. Y. Ts'o. btrfs: Inconsistent behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77340.html, Apr 2018.

[58] UTSASLab. Crash-consistency bugs studied and reproduced. https://github.com/utsaslab/crashmonkey/blob/master/reproducedBugs.md.

[59] UTSASLab. New crash-consistency bugs found. https://github.com/utsaslab/crashmonkey/blob/master/newBugs.md.

[60] J. Verge. Internap Data Center Outage Takes Down Livestream And Stackexchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/, 2014.

[61] R. S. V. Wolffradt. 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, 2014.

[62] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled io stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 211–226, Berkeley, CA, USA, 2018. USENIX Association.

[63] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, Nov. 2006.

[64] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors (awarded best paper!). In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 273–288. USENIX Association, 2004.

[65] C. Yu. f2fs: keep isize once block is reserved cross eof. https://sourceforge.net/p/linux-f2fs/mailman/message/36104201/, Nov 2017.

[66] C. Yu. f2fs: enforce fsync_mode=strict for renamed directory. https://lkml.org/lkml/2018/4/25/674, Apr 2018.

[67] C. Yu. f2fs: fix to set keep_size bit in f2fs_zero_range. https://lore.kernel.org/patchwork/patch/889955/, Feb 2018.

[68] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 1–14, Santa Clara, CA, 2016. USENIX Association.

[69] S. Zhang, H. Catanese, and A. A.-I. Wang. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 15–22, Santa Clara, CA, 2016. USENIX Association.

[70] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, 2014. USENIX Association.