Crash Consistency Test Generation for the Linux Kernel

by

Arvind Raghavan

THESIS

Presented to the Faculty of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

B.S. of Computer Science, Turing Scholar

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Crash Consistency Test Generation for the Linux Kernel

Arvind Raghavan
The University of Texas at Austin, 2020

Supervisor: Vijay Chidambaram

Modern file systems try very hard to ensure that they can recover correctly after a crash. However, the complexity of file systems and the large space of possible bug-triggering workloads make this difficult to achieve. To find bugs, UT has built Crashmonkey, an in-house test framework that can efficiently simulate various crash scenarios, and the Automatic Crash Explorer (ACE), that can exhaustively generate workloads for Crashmonkey to test [1]. The work in this paper builds on both frameworks by creating an adapter to port ACE workloads to `xfstests` [2], the Linux kernel filesystem test suite. In addition, I upgrade ACE to programmatically generate crash consistency `xfstests` tests aimed at code coverage in the Linux kernel. I am in the process of patching 9 of these generated tests into `xfstests`. Finally, I present a fuzzer that builds on the ACE framework to look for new bugs.

Table of Contents

Chapter 1

Introduction

One of the main goals of file systems is to recover correctly after a crash. Here, correctness is typically defined in terms of consistency. A file system is in a consistent state after a crash if its state could be arrived at by replaying the operations in order and stopping after any fully completed operation. Common examples of inconsistency include operations being out-of-order (that is, the state of the file system reflects the completion of operation B, but not operation A, even though A occurs before B in the workload) and operations being partially committed (for example, only part of a single write appears on disk).

One of the main reasons crash consistency is difficult to achieve is because there are several layers of indirection involved in writing to a file. Because writing to disk is an expensive operation, the operating system will buffer several writes in memory and "flush" them to disk at once. This boosts performance, but leaves the user in the unfortunate situation of calling "write", but not actually knowing whether their data is written to disk. Furthermore, even after writes are issued and arrive at a memory device, some of these devices store data in a cache before finally writing it to stable storage. If power loss occurs at any point before the data reaches stable storage, data may be lost.

There are many techniques, including journaling and soft updates, which modern file systems use to ensure they can quickly return to a consistent state after a crash. However, these techniques don't tell the user which files will be persisted, only that the files that are persisted will not be inconsistent. In order to guarantee that specific files are persisted in a consistent state, file systems provide system calls such as `sync` and `fsync`, which guarantee that the user's writes to specific files have been written to stable storage upon returning. However, these additions add layers of complexity to file system implementations and touch basically all parts of the code. Thus, any new changes to file systems have to be carefully reviewed to ensure they don't break consistency guarantees.

Given this, file system developers find it useful to have regression tests, which are functional tests they can re-run upon each update to ensure that file system behavior remains correct. However, for various reasons, including the fact that crash consistency tests have historically had to power cycle a machine (or virtual machine) and were thus slow, crash consistency tests are largely absent from these regression test repositories. Thus, developers have to wait until things break in production to discover bugs.

Crashmonkey and the Automatic Crash Explorer (ACE) [1] solve part of this problem by systematically searching for crash consistency bugs. Crashmonkey is a flexible test harness that can take a workload, consisting of C++ code that implements a "test template" class, and determine whether or not it causes a crash consistency bug. ACE is a workload generator that can take a set of operations and exhaustively generate all possible length-n workloads for Crashmonkey to check. It does this by first generating workloads in a high-level "pseudocode" language called J-Lang, which will be explained in detail later, and then using an adapter to convert those workloads into Crashmonkey workloads in C++. By using ACE to generate all possible workloads of lengths 1, 2, and 3, and running those workloads with Crashmonkey, the UT Systems and Storage Lab have found ten bugs [3] across btrfs [4] and F2FS [5], and even a bug in the formally verified FSCQ [6]. These results show that crash consistency bugs are widespread and that even techniques such as formal verification cannot adequately guarantee crash consistency.

I present three main contributions which aim to augment Crashmonkey and ACE in order to find and prevent new file system bugs. The first contribution is an `xfstests` adapter for ACE, which can convert single workloads generated by ACE into the format used by `xfstests`, the Linux file system regression test suite. Because Crashmonkey's C++ workloads and ACE's high-level J-Lang workloads differ in format from the Bash script tests in `xfstests`, bugs found by Crashmonkey and ACE have to be informally described when they are reported [7, 8], instead of being submitted with a test highlighting the bug. While this still allows Linux developers to fix the bugs, it means that the already busy kernel developers have to handwrite tests for these bugs if they wish to add them to the regression test suite. The adapter addresses this by making it easy to convert any future bugs found into `xfstest` tests. These tests can be sent as patches to the Linux developers to ensure that the bugs don't re-emerge.

The second contribution is upgrading ACE to automatically generate coverage tests for crash consistency bugs in the `xfstests` format. Currently, Crashmonkey and ACE require that bugs occur in production before they are found. In order to prevent these bugs from entering production in the first place, it is necessary to have crash consistency regression tests with comprehensive code coverage. Linux developers can use these tests to ensure that their new releases don't break any crash consistency guarantees. The automatic test generator that I present has created nine `xfstests` tests, each centered around single operation (i.e. `rename`, `falloc`, `write`, etc.). The tests themselves are each composed of around 30 tests for different crash scenarios. The Linux developers have agreed that these generated tests provide valuable code coverage [9], and I am currently in the process of getting the generated `rename` test patched [10] into `xfstests`. I plan to follow up that patch with the remaining eight tests.

The final contribution is a fuzzer that builds on the ACE framework to find bugs in longer workloads. One of the drawbacks of using ACE is that exhaustively searching the space of workloads is not feasible for longer workload lengths. However, there exist bugs in longer workload, and searching around the space of previously patched bugs may be valuable, as there could exist similar, unpatched bugs. Fuzzing has proven to be a useful technique for search for bugs in programs with large input spaces [11]. I present a fuzzer that can, given a workload "starting point", search a bounded space around that workload for similar bugs. I ran the fuzzer on the smaller, patched bugs initially found by Crashmonkey [3] to see if any similar bugs remained after the patches. The fuzzer uncovered some strange behavior in `btrfs` that I reported to the Linux kernel developers [12]. While the behavior was ultimately determined to not be unexpected, the ability of the fuzzer to find such behavior even in smaller workloads highlights the merit of this approach. I plan to run this fuzzer on longer workloads in order to look for more bugs.

Chapter 2

Background

The contributions presented build on the frameworks created by Crashmonkey and the Automatic Crash Explorer (ACE) [1]. Thus, a clear understanding of how both Crashmonkey and ACE work is essential to understanding the work presented in this paper.

2.1 Crashmonkey

Crashmonkey workloads are expressed as C++ programs with a setup and a run phase. The setup phase is used to setup an initial, "good" disk state, and the run phase contains a set of operations that potentially trigger a bug followed by a function call that simulates a crash. Crashmonkey tests the workload by setting up the disk as expressed in the setup phase, running the operations in the run phase, and then logging all the block writes to disk that the file system produces. When the workload reaches the "crash point", instead of actually simulating the power failure, Crashmonkey builds a set of potential "restored" disk states to verify by randomly dropping a subset of "in-flight" block writes.

It is important to note that these block writes are often tagged with flags that specify behavior wanted from the device. Among these flags are the Forced Unit Access (FUA) flag, which indicates that the request should not return until the data has been written to disk, and the flush flag, which indicates the device's cache must be flushed. Crashmonkey ensures that the subset of block writes that it drops is consistent with the flags specified; thus, it only creates disk states that could actually be observed on a device after a power loss. By choosing to drop a subset of writes instead of all "in-flight" writes, Crashmonkey has the flexibility to programmatically generate disk states that would be difficult to encounter by randomly simulating power loss.

Finally, Crashmonkey verifies whether or not disk states are consistent by keeping track of which files are supposed to be synced. For each of the synced files, it ensures that the file data and metadata of the file is persisted in each of the potential reconstructed states.

2.2 Automatic Crash Explorer (ACE)

ACE generates workloads by bounding the space of workloads in three dimensions: the length of the workload, the set of operations, and the set of files to operate on. It then exhaustively generates all possible workloads within these bounds. Because Crashmonkey workloads are defined as C++ programs, ACE chooses to define a new, high-level language, called J-Lang, for which it is easier to programmatically generate workloads. It then uses an adapter to convert those J-Lang workloads into Crashmonkey workloads in C++. The `xfstests` adapter presented in Chapter 3 will build on this framework by converting J-Lang workloads into the `xfstest` format.

2.2.1 J-Lang Format

A sample J-Lang workload is shown in Listing 2.1.

Listing 2.1: Sample J-Lang workload that checks if a write persists

```
# setup
mkdir A
sync

# run
create A/foo
write A/foo overlap_start
fsync A/foo
*crash*
```

J-Lang workloads consist of a setup and a run phase, which mirror the setup and run phases that Crashmonkey uses. For the most part, operations in the J-Lang language can be directly translated into shell/C++ commands. The only exceptions to this occur with syscalls that require an integer offset and length, such as the `write` shown in Listing 2.1 and the `falloc` syscall which allocates space in a file. Iterating over all possible offsets and lengths would be very costly and would likely not add any meaningful coverage. Thus, J-Lang defines six types of `ranges` which have coverage over the space of page alignment and current file length. Table 2.1 shows the types of `ranges` and their translations. For operations that take range arguments, the Crashmonkey adapter translates the range parameter to offset and length values when creating the C++ code. Note that this translation depends on the file size, so the Crashmonkey adapter keeps track of file sizes for all open files when translating the J-Lang file.

Table 2.1: Range conversions to offset and size

range	offset	size (bytes)
append	EOF	32768
overlap_unaligned_start	0	5000
overlap_unaligned_end	min(0, EOF - 5000)	5000
overlap_start	0	8192
overlap_end	min(0, EOF - 8192)	8192
overlap_extend	min(0, EOF - 2000)	5000

2.3 Sync Operations

Finally, it is important to note that Crashmonkey focuses specifically on finding bugs related to the `sync`, `fsync`, and `fdatasync` system calls, as these are the main primitives given to user applications to ensure their data is saved in stable storage. File systems don't guarantee that any data is saved to disk unless one of these syncing operations is called. Thus, the workloads Crashmonkey evaluates end with a call to a syncing operation followed by a crash. There exists a bug if a file or its metadata is not persisted even after an explicit sync.

Furthermore, while many file systems provide stronger different guarantees, Crashmonkey and ACE only enforce the `POSIX` standard, which is the baseline set of guarantees that all file systems must implement. According to these standards, an `fsync` of a file should persist its data and metadata, an `fdatasync` is only guaranteed to persist a file's data, and a `sync` persists all modifications.

Chapter 3

xfstests Adapter

The `xfstests` Adapter converts J-Lang workloads that ACE generates into `xfstest` tests.

3.1 The `xfstest` template

Tests in `xfstests` are written as Bash scripts. `xfstests` provides a template for handwriting new tests which mainly consists of boilerplate involving mounting the test drive, formatting test output, etc. The adapter relies on the following modifications to the template in order to support programmatic test generation.

1. `dm-flakey`

   `xfstests` includes library functions that use `dm-flakey` [13], a device mapper that can simulate power loss by dropping all reads and writes not committed to disk. By sourcing the file `common/dmflakey` in the `xfstest` repo, an `xfstests` script gets access to the `_init_flakey`, `_flakey_drop_and_remount`, and `_cleanup_flakey` operations. These operations, create the flakey device, simulate power loss, and cleanup the flakey device respectively.

2. `general_stat`

   In order to ensure that the filesystem is fulfilling its guarantees, the adapter must know what data and metadata is guaranteed to be persisted. However, the data and metadata that should be saved differs depending on whether the file synced is a regular file or a directory and whether the file is `fsync`ed or `fdatasync`ed (a call to `sync` is equivalent to calling `fsync` on all open files). I define a helper function, `general_stat`, which prints out the data and metadata that should be persisted based on the file type and the sync operation. Table 2 shows the shell commands used by `general_stat` to extract data. Note that `stat <file>` prints out the metadata of a file, `md5sum <file>` prints out a hash of the file data, and `ls <directory>` lists the directory contents.

Table 3.1: Ouptut of `general_stat`

File Type	Sync Type	Output
Regular File	`fsync`	`stat $file` `md5sum $file`
Regular File	`fdatasync`	`md5sum $file`
Directory	`fsync`	`stat $dir` `ls $dir`
Directory	`fdatasync`	`ls $dir`

3. `check_consistency`

   Using the `dm-flakey` device and the `general_stat` helper function described above, I define a helper function that checks the consistency of a set of files by saving the state of the files, simulating power loss, and ensuring that the restored state is equivalent to the saved state. Listing 3.1 contains the pseudocode of the `check_consistency` helper function simplified to only handle a single file.

Listing 3.1: `check_consistency function`

```
check_consistency()
{
    file="$1"    # first arg is file
    sync_op="$2" # second arg is fsync/fdatasync

        # get initial state
        before=$(general_stat $file $sync_op)

        # simulate power loss
        _flakey_drop_and_remount

        # get restored state
        after=$(general_stat $file $sync_op)

        if [ "$before" != "$after" ]; then
                # fail
        fi
}
```

3.2 Generating Tests

There are two main phases that the `xfstests` adapter undergoes when converting a J-Lang workload. The first is the translation phase, which simply translates the operations in the J-Lang file to the `xfstest` equivalents. Listings 3.2 and 3.3 show a sample J-Lang workload and the relevant translated code that the adapter inserts into the template. Because the J-Lang language resemble Bash, the tests appear similar, with some exceptions. Namely, in `xfstests` tests, all files are referenced relative to the test directory, `$SCRATCH_MNT`, and operations such as writing to files and syncing require library helper functions.

Note that the `append` operation gets translated to an offset and range. Like the Crashmonkey adapter, the `xfstests` adapter converts the intermediate `range` parameter according to Table 2.1. In addition, note that operations that ACE generates potentially have multiple dependencies. For example, the dependencies for writing to a file include that the file must exist, its parent directory must exist, and the file must have the correct read/write permissions. However, a workload with multiple writes shouldn't try to create the parent directory multiple times. Thus, during translation, the adapter keeps track of the current disk state and only inserts instructions to fill unsatisfied dependencies.

Listing 3.2: Basic J-Lang workload

```
# run
mkdir A 0777
write Afoo append
fsync A
*crash*
```

Listing 3.3: Listing 3.2 converted to `xfstests`

```
# run
mkdir $SCRATCH_MNT/A -p -m 0777
touch $SCRATCH_MNT/A/foo
 _pwrite_byte 0x22 0 32768 $SCRATCH_MNT/A/foo ""
$XFS_IO_PROG -c "fsync" $SCRATCH_MNT/A
check_consistency <some files>
```

The second phase is the sync phase, in which the adapter determines which files should be synced and thus passed to `check_consistency` for verification. Determining which files should be synced is simple for workloads that end with a single sync operation followed by a crash, like the one shown in Listing 3.3. However, determining which files are synced becomes more difficult for longer workloads that, say, sync some subset of files, modify another subset, sync again, and so on. To determine which files are synced, the adapter must keep track of some state.

While translating the J-Lang file, the `xfstest` adapter maintains state consisting of the set of open files, the set of synced files, and the set of data-synced files. The latter is important for the previously mentioned `fdatasync` operation, which persists file data but doesn't make any guarantees for file metadata. Upon encountering an operation that opens a file, the adapter adds it to the open set. Every `fsync` or `fdatasync` operation moves the corresponding file into the synced files and data-synced file sets accordingly. Conversely, every write or metadata-changing operation removes the files from the synced sets if they are present. Finally, a `sync` operation, which flushes all data to disk, should move every file in the open set to the synced set. Keeping track of this state allows the adapter to determine which files should be synced at the end of the workload, and thus which files it should pass to `check_consistency`. Using these two phases, the adapter is able to convert J-Lang workloads directly to `xfstests` tests.

Chapter 4

Automatic Coverage Test Generation

When the original Crashmonkey paper [1] was published, the authors added a patch to `xfstests` containing a test that combined 37 “hard link” related tests into a single test [14]. This test was made without the `xfstest` adapter and had to be hand constructed. The Linux developers expressed interest [9] in having more of these “concise” test cases generated for different operations (`rename`, `truncate`, etc.), as this would add much needed code coverage for crash consistency in the kernel. I wished to programmatically generate these tests so that they could be created for a number of different operations. To do this, I agumented ACE to generate workloads in a new high-level language specification, J-Lang V2, which combines several J-Lang workloads into a single test. I then upgraded the `xfstest` adapter from Chapter 3 to support converting the new J-Lang V2 workloads into single `xfstests` tests with code coverage for many different crash scenarios.

4.1 The J-Lang V2 Format

The J-Lang V2 format consists of two sections. The first section consists of lines prefixed with ”file” and ”option”, and describes variables that can take on multiple values. The remainder of the file is a "template", which consists of a standard J-Lang workload that uses those variables. The J-Lang V2 workload thus describes a test that runs the "template" workload multiple times for every combination of the "file" and "option" variables. An example of a simple write workload is shown in Listing 4.1.

Listing 4.1: An Example J-Lang V2 workload

```
# J2-Lang
file1 foo A/foo
option1 append overlap_unaligned_start overlap_extend
write $file1 $option1
```

Notice that, unlike the original J-Lang format, the J-Lang V2 format does not include a setup section. This is because the coverage tests that J-Lang V2 files describe are meant to be simple in nature, as the tests will be run repeatedly with different files and options. The setup section is more useful for single tests that are trying to exhibit more complex behavior. Thus, the operations in J-Lang V2 are analogous to operations in the "run" section of J-Lang workloads.

In addition, notice that the sync operation is left out of Listing 4.1. Because this workload contains multiple tests, each with different combinations of files, the adapter cannot statically determine which files should be synced. This is because each test in a J-Lang V2 workload may operate on different files, and thus a different subset of files may be synced at the end of each test. While it would be possible to have infrastructure at run-time that tracks which files are synced, this would complexity to the tests and make it more difficult for the kernel developers to reason about the output if the test failed.

Thus, I decided to have a single sync operation at the end of each workload, which significantly clarifies the test output. The adapter inserts code that inserts a single `sync`, `fsync`, or `fdatasync` operation on one of the open files or its parent directory and pass it to `check_consistency`. Thus, for the workload shown in Listing 4.1, there are five different possible calls to sync for each value of `file1` and `option1`. They consist of a single call to `sync` and four total calls of `fsync` or `fdatasync` on either `file1` or its parent directory. Thus, the workload contains 30 different tests (2 files * 3 write options * 5 sync options).

Because ACE does an exhaustive search, it is already iterating through tests in the same fashion that J-Lang V2 workloads describe. In particular, when ACE is directed to create 1-length tests, it will generate 30 separate J-Lang tests for the `write` operation, each testing a single case of the workload shown in Listing 4.1. Thus, modifying ACE to output tests in this format was relatively straightforward; instead of making ACE iterate over several values for different variables, it instead outputs those variables and their values into the J-Lang V2.

4.2 Changing the `xfstest` Template

Notice that the J-Lang V2 workload in Listing 4.1 contains the `append`, `overlap_unaligned_start`, and `overlap_extend` options. J-Lang V2 workloads share the same `range` abstraction as the J-Lang workloads. However, converting these ranges to the values specified by Table 2.1 is trickier to implement for J-Lang V2 tests. When given a sinlge test, the adapter could directly convert the append instruction to an offset and size for a write in the translation phase. However, the offset and size cannot be known statically for J-Lang V2 workloads. For example, consider the workload that consists of two consecutive writes, both operating on a set of files including foo. In the case that the second append writes to foo, its offset depends on whether or not the first append also writes to foo. Thus, the calculation of offset and size must be done at run-time.

To account for this, I added a helper function, `translate_range`, to handle the range translation at run-time. A simplified version of this helper function is shown in Listing 4.2.

4.3 Generating Coverage Tests

In order to generate these coverage tests, I modified the `xfstest` adapter to support J-Lang V2 files. To generate an `xfstests` test from a J-Lang V2 file, the adapter defines a workload function that is parameterized by the variables described in the J-Lang V2 file. Thus, the adapter generates a test that iterates over all possible values for those variables, runs the workload function, and ensures that the resulting disk state is crash consistent. Listing 4.3 shows the code generated by the `xfstests` adapter when given the J-Lang V2 file in Listing 4.1 as input.

The generated code is able to iterate over all possible combinations of variables and run a consistency checking test. Note the calculation of the `uniques` variable at the end of the code, which represents the run-time calculation of which files to sync. The `xfstest` adapter first creates the variable `fsync_files`, which contains all of the files used in the current iteration and their parent directories. As some of these values can overlap (say if two files share the same parent directory), the `uniques` variable filters out the duplicates. Then, the workload template function is called with all of these potential files to sync and all possible sync operations (omitting `sync` for simplicity). Thus, the adapter is able to generate code that can try to sync all possible combinations of files at run-time.

Using this adapter, I have been able to build concise tests for nine operations (`rename`, `falloc`, `write`, `dwrite`, `mmapwrite`, `unlink`, `fsetxattr`, `removexattr`, and `truncate`). I sent out the rename test as patch [10] to `xfstests`, and it is currently in the process of getting reviewed and accepted. I intend to follow up by submitting patches for the remaining eight operations.

Listing 4.2: The `translate_range` helper function

```
translate_range() {
        # get current file size
        size=$(stat -c %s $1)

        # match the 'range' parameter and return
        # the appropriate offset and length
        case $2 in
                append)
                        offset=$size
                        length=$((offset + 32768))
                ;;
                overlap_unaligned_start)
                        offset=0
                        length=5000
                ;;
                # ...
        esac
        echo $offset $length
}
```

Listing 4.3: The converted `xfstest` test for Listing 4.1

```

# workload template function
function write_template() {
    local file1="$1"
    local option1="$2"
    local file_to_sync="$3"
    local sync_op="$4"

    _mount_flakey

    # basic workload from J-Lang V2 file
    mkdir -p $(dirname $file1)
    touch $file1
    range=$(translate_range $file1 $option1)
    _pwrite_byte 0x22 $range $file1
```

```
    # sync a file/dir and ensure its data persists
    $XFS_IO_PROG -c "$sync_op" $file_to_sync
    check_consistency $file_to_sync $sync_op
    clean_dir
}

# variables defined in J-Lang V2 file
file1_options=(
    "$SCRATCH_MNT/A/foo"
    "$SCRATCH_MNT/foo"
)

option1_options=(
    "append"
    "overlap_extend"
    "overlap_unaligned_start"
)

# iterate over all options
for f1 in ${file1_options[@]}; do
  for opt1 in ${option1_options[@]}; do
    # find unique open files to sync
    fsync_files=("$f1" "$(dirname $f1)")
    uniques=($(for v in ${fsync_files[@]}; do
      echo $v;
    done | sort -u))

    for sync_file in ${uniques[@]}; do
      for sync_op in fsync fdatasync; do
        write_template $f1 $opt1 $sync_file $sync_op
      done
    done
  done
done
```

Chapter 5

Fuzzer

One of the drawbacks of using ACE is that exhaustively searching the space of workloads is not feasible for longer workload lengths. Fuzzing has been used to effectively find bugs in programs with large input spaces [11], and has been shown to be specifically useful in finding crash consistency bugs [15]. In addition, searching the space around previously patched bugs could yield new bugs, because file systems are complex enough that if a bug exists, there are also likely to exist many similar bugs. Thus, I built a fuzzer that extends the ACE framework in order to search around the space of previously patched, longer bugs in order to find new bugs.

5.1 Approach

Consider, for example, a "starting point" workload with n operations. One can imagine searching the space of workloads around this starting point by changing the options passed to each operation. One could also imagine swapping the files that each operation acts on or even interchanging operations altogether. We can define any of those changes to be a mutation.

Now, let the set of length-1 mutations from a given starting point be the set of all workloads that can be reached from the starting point by making a single mutation to a single operation in the workload. The set of length-2 mutations would be the set of workloads reachable by mutating two operations, and so on. If we extend the set of mutations to include changing options, files, and operations altogether, then calculating the set of length-n mutations generalizes to an exhaustive search, which is what ACE performs. While this exhaustive search cannot be performed for long workloads, defining these mutation sets effectively narrows the search space.

I built a basic fuzzer that, given a "starting point" workload, computes the set of lenghth-1 mutations. To bound the search space, I limit the set of mutations to include changing files, options, and interchanging related operations (such as swapping `fsync` for `sync`). The pseudocode for calculating length-1 mutations is shown in Listing 5.1. For each workload passed to `add_fuzzed_workload`, the fuzzer creates an output J-Lang file, which is run through Crashmonkey adapter to create a Crashmonkey workload.

Listing 5.1: Fuzzer Pseudocode

```
for i in length(workload):
    instruction = workload[i]

    for mut in mutations(instructions):
        new_workload = copy(workload)
        new_workload[i] = mut

        add_fuzzed_workload(new_workload)
```

5.2 Results

As a preliminary test, I ran this fuzzer on the set of 11 bugs that Crashmonkey previously found [3] to see if any similar bugs still existed after the patches. While I didn't find any new violation of POSIX standards, I did find some strange behavior on `btrfs`. The workload that triggered the strange behavior is shown in Listing 5.2.

Listing 5.2: Workload triggering strange btrfs behavior

```
mkdir A
mkdir B
mkdir A/C
creat B/foo
sync
link B/foo A/C/foo
*crash*
```

Upon recovering from a crash, the link `A/C/foo` was not persisted. This is expected, because the `sync` appears before the link creation. However, replacing the `sync` with an `fsync B/foo`, which should not have stronger guarantees, causes the link to persist after the crash. While `POSIX` standards make no guarantees that the link should persist after the crash, it is strange that `fsync`, a weaker operation than `sync`, had a stronger effect. We believed it may be indicative of some underlying bug in the file system. I reported [12] this behavior to developers. While they ultimately determined that this behavior was not unexpected, the fuzzer's ability to find this sort of behavior shows promise for finding future bugs. In the future, I plan to run this fuzzer on other longer previously found bugs.

Conclusion

As file systems continue to become more complex, kernel developers will increasingly rely on automated crash consistency checkers, such as ACE and Crashmonkey, and regression test suites, such as `xfstests`, to help ensure correctness. By building tools for portability between Crashmonkey and `xfstests` and submitting test patches, I have been able to help extend the Linux regression test suite to add much needed crash consistency tests. In addition, the adapter infrastructure as well as the fuzzing framework that I have built will allow the UT Systems and Storage Lab to find new bugs and more easily integrate test cases for them into the Linux development cycle.

Acknowledgments

Firstly, I would like to thank my supervisor, Vijay Chidambaram, for his openness to let me do exciting research as an undergrad. His encouragement helped motivate me to accomplish much more than I thought I could. I would also like to give a huge thanks to Jayshree Mohan. Without her responses to my unending questions and emails, I wouldn't have been able to complete this thesis.

Bibliography

[1] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey and ACE: Systematically Testing File-System Crash Consistency. *ACM Transactions on Storage (TOS)*, 15(2):1–34, 2019.

[2] Jonathan Corbet. Toward better testing. `https://lwn.net/Articles/591985/`, March 2014. [Online; posted 26-March-2014].

[3] UTSASLab. New Crash-Consistency Bugs Found. `https://github.com/utsaslab/crashmonkey/blob/master/newBugs.md`, 2018.

[4] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[5] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 273–286, 2015.

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

[7] Jayshree Mohan. btrfs: Hard Link Not Persisted On fsync. https://www.spinics.net/lists/linux-btrfs/msg76878.html, 2018.

[8] Jayshree Mohan. btrfs: Inconsistent Behavior of fsync in btrfs. https://www.spinics.net/lists/linux-btrfs/msg77219.html, 2018.

[9] Arvind Raghavan. xfstests: Additional Crashmonkey Tests. https://www.spinics.net/lists/fstests/msg13920.html, 2020.

[10] Arvind Raghavan. fstest: Crashmonkey rename tests ported to xfstests. https://patchwork.kernel.org/patch/11508647/, 2020.

[11] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl, 2019.

[12] Arvind Raghavan. Strange sync/fsync behavior in btrfs. https://www.spinics.net/lists/fstests/msg13758.html, 2020.

[13] dm-flakey. https://www.kernel.org/doc/Documentation/device-mapper/dm-flakey.txt.

[14] Jayashree Mohan. fstest: Crashmonkey tests ported to xfstest. https://patchwork.kernel.org/patch/10679551/, 2018.

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