Enhanced File System Testing through Input and Output Coverage

Yifei Liu
Stony Brook University
Stony Brook, NY, USA
yifeliu@cs.stonybrook.edu

Geoff Kuenning
Harvey Mudd College
Claremont, CA, USA
geoff@cs.hmc.edu

Md. Kamal Parvez
Stony Brook University
Stony Brook, NY, USA
mparvez@cs.stonybrook.edu

Scott A. Smolka
Stony Brook University
Stony Brook, NY, USA
sas@cs.stonybrook.edu

Erez Zadok
Stony Brook University
Stony Brook, NY, USA
ezk@cs.stonybrook.edu

Abstract

Effective file system testing relies on coverage to detect bugs and enhance reliability. We analyzed real file system bugs and found a weak correlation between code coverage, the most commonly used metric, and test effectiveness; many bugs were in covered code but remained undetected. Our study also showed that covering diverse file system inputs and outputs—system call arguments and return values—can be key to detecting the majority of observed bugs.

We present *input coverage* and *output coverage* as new metrics for evaluating and improving file system testing, and have developed the IOCov framework for computing these metrics. Unlike existing system call tracers, IOCov computes coverage using only the calls relevant to testing, excluding unrelated ones that should not be counted. To demonstrate IOCov's utility, we used it to extend the existing testing tool CrashMonkey into CM-IOCov, which achieves broader input coverage and more thorough detection of crash consistency bugs. Our experimental evaluation shows that IOCov computes input and output coverage accurately with minimal overhead. IOCov is applicable to different types of file system testing and can provide insights for improvement as well as identify untested cases based on coverage results. Moreover, the bugs found exclusively by CM-IOCov are 2.1 and 12.9 times more than those found exclusively by CrashMonkey on the 6.12 and 5.6 kernels, respectively, demonstrating the effectiveness of the IOCov-based coverage approach.

SYSTOR '25, Virtual, Israel
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2119-9/2025/09
https://doi.org/10.1145/3757347.3759138

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **File systems management**.

Keywords: File System Testing, Input Coverage, Output Coverage, Crash Consistency

1 Introduction

File systems serve as the backbone of modern storage [19, 20, 48], supporting numerous applications such as databases [13], cloud platforms [52], and big data analytics [49, 50, 90]. Given their critical role, file system bugs pose significant risks to overall system reliability [29, 56], including data loss, data corruption, and system crashes. Consequently, various testing techniques have been developed to detect file system bugs and improve system reliability [44–46, 58, 83]. File system testing, however, remains a challenge due to the complexity of file systems and their stringent requirements, such as data integrity, fault tolerance, and POSIX compliance [66]. Despite the availability of a number of testing tools, such bugs continue to emerge on a regular basis [38], indicating that existing testing methods are inadequate and there is room for improvement.

Various *coverage metrics* have been proposed based on specific testing approaches [47]. For example, regression testing [57, 68] seeks to achieve functionality coverage, while model checking targets state coverage [46, 73]. *Code coverage* is the most widely used metric in file system testing [38]. However, the effectiveness of code coverage for file systems remains insufficiently studied. It is still unclear whether increased code coverage leads to identifying more bugs. Additionally, even when developers know which code segments are not covered, modifying tests to enhance coverage is a challenge due to the complexity of file system code [4, 23].

Most existing analyses of code coverage effectiveness [28, 33] focus on small user applications rather than large, low-level systems like file systems. Furthermore, no coverage metrics have been specifically designed for file system testing to help developers improve testing and detect more bugs.

Our Contributions. This paper addresses the following unique challenges: (1) conduct a practical bug study to evaluate the effectiveness of code coverage in file system testing; (2) design coverage metrics tailored for file system testing that are both effective and developer-friendly; and (3) leverage these metrics to improve testing and uncover more crash consistency bugs.

We first conducted an analysis of recent file system bugs that led to the discovery of a weak correlation between code coverage and test effectiveness. In terms of triggering file system bugs, we then identified the importance of covering both (a) diverse test inputs, including system calls (syscalls) and their arguments, and (b) test outputs, such as syscall returns and errors. The majority of these bugs require specific inputs to be triggered and typically occur along an exit path, which affects the output. Hence, we define *input and output coverage* as criteria for evaluating and improving file system testing tools. We partition the input and output spaces according to syscall argument and return types, and measure input and output coverage by analyzing the frequency of segments exercised by testing tools.

Computing input and output coverage involves tracing tested syscalls while excluding unrelated noise that is not part of the test workload. Because existing syscall tracers [1] cannot solely focus on tracing file system calls, we designed and implemented **IOCov** to compute input and output coverage for testing tools. We applied IOCov to various file system testing tools, including black-box testing, regression testing, fuzzing, and model checking, uncovering untested input/output partitions in all of them and gaining insights into how they can be improved.

To demonstrate the utility of IOCov, we enhanced the crash consistency testing tool CrashMonkey [58] by 1) significantly improving its input coverage while keeping the rest of the system unchanged, and 2) having it run IOCov. The improvement in input coverage comes from a driver that provides more diverse syscall arguments (*i.e.*, inputs) than the original CrashMonkey. We refer to this new version of CrashMonkey as **CM-IOCov**. We compared CM-IOCov to the original CrashMonkey in terms of the ability to detect crash-consistency bugs in the Btrfs file system [67], and found that CM-IOCov identified 74.1% more test failures (potential bugs) than the unmodified CrashMonkey on the new Linux 6.12 kernel.

In summary, this paper makes the following contributions:

1. By using xfstests to analyze real bugs, we revealed the limitations of code coverage (it often misses bugs even in covered code) and highlighted the importance of covering diverse syscall inputs and outputs.
2. We formalized *input and output coverage*, allowing us to evaluate and improve file system testing by partitioning input and output spaces, thereby addressing the limitations of code coverage.
3. We designed and implemented IOCov to evaluate the input and output coverage of file system testing tools. We applied IOCov to a number of testing tools, in the process deriving insights for their improvement.
4. We created CM-IOCov to enhance crash-consistency testing (*i.e.*, CrashMonkey). CM-IOCov detects more bugs than CrashMonkey and demonstrates the effectiveness of input coverage in real-world bug detection.

To promote reproducibility and future research, we have open-sourced both IOCov and CM-IOCov at: *https://github.com/sbu-fsl/IOCov* and *https://github.com/sbu-fsl/CM-IOCov*. The rest of this paper is organized as follows. Section 2 considers the effectiveness (or lack thereof) of code coverage when it comes to bug detection, and underscores the role inputs and outputs can play here. Section 3 defines input/output coverage, and presents the design and implementation of the IOCov framework. Section 4 focuses on the role CM-IOCov plays in improving crash consistency testing. Section 6 presents our experimental results, Section 7 discusses related work, and Section 8 offers our concluding remarks.

2 File System Bug Study

This section addresses two questions: (1) whether code coverage is effective for file system testing, and (2) which aspects of testing are crucial for detecting bugs. To answer these questions, we analyzed recent file system bugs which led us to devise *input and output coverage criteria* for file system testing. Unlike previous file system bug studies that focused on bug patterns and classification [51, 87], our study not only examined the effectiveness (or lack thereof) of code coverage in finding bugs, but also identified the key factors that contribute to bug detection.

2.1 Code Coverage in FS Testing

A common approach to assessing code coverage effectiveness is *mutation testing*, which involves introducing small faults and checking whether test suites with increased code coverage can detect more faults than those with lower coverage [28, 33]. This method, however, is not applicable to in-kernel file systems, where even small faults can lead to serious OS errors, make the file system unmountable, or damage basic file utilities, preventing us from executing any further tests. As a result, we adopt a different approach to evaluating the correlation between code coverage and bug detection by studying known file system bugs [30, 39]. If covering the buggy code region leads to bug detection, this indicates a correlation between code coverage and the test's effectiveness in exposing the bug [39].

Developers identify and resolve Linux kernel file system bugs by submitting *patches*, which, after review, are merged into the kernel repository [35]. Therefore, analyzing *accepted* patches in the form of Git commits can reveal information about previously buggy file system code [74]. We collected the 100 most recent Git commits [75] from 2022 for each of the two popular Linux file systems, ext4 [55] and Btrfs [67], amounting to 200 commits in total. These were the latest commits available when we began this project. Some commits were not bug fixes; instead they introduced new features, performance optimizations, or maintenance changes [51].

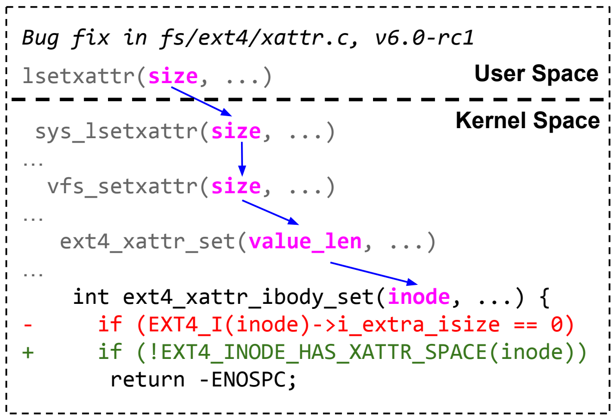
We then applied Lu *et al.*'s taxonomy [51] to identify bug-fix commits, finding 51 ext4 bugs and 19 Btrfs bugs. (The lower count for Btrfs is due to major code refactoring in December 2022.) These 70 bugs span all four file system bug categories of Lu *et al.*'s taxonomy: 37 semantic, 3 concurrency, 20 memory, and 10 error-code bugs. Next, we ran xfstests, a widely used test suite, on ext4 and Btrfs using Linux kernel v6.0.6, the latest version in which the extracted bugs remained unfixed. We executed all generic and file system-specific tests and used `Gcov` [32] to measure line, function, and branch coverage of the file system sources.

For each bug fix, we inspected the `Gcov` reports to determine if xfstests covered the pertinent code, and then reviewed the test logs and commit messages to determine if the suite detected the bug. Our aim was to assess code coverage effectiveness at different levels (line, function, and branch) to determine if xfstests could detect bugs in the covered code. To ensure accuracy, two individuals with file system expertise independently cross-validated the results.

The results of our study showed that xfstests failed to detect any of the 70 bugs even though it covered many code segments related to these bugs. Specifically, xfstests covered relevant lines of code for 37 of the 70 bugs (53%) but did not detect those bugs, indicating that line coverage does not ensure bug detection. Additionally, it covered the functions of 43 of the 70 bugs (61%) and the branches of 20 of the 70 bugs (29%) without detecting the bugs. Consequently, all three code-coverage metrics reveal that merely covering code does not mean bugs that lie within it will be detected. Worse, among all of the bugs that we studied, xfstests executed each buggy line of code an average of over 13.8 million times per bug, but remained unable to uncover the bug concealed within those lines. This indicates that repeatedly covering code may not be useful for bug detection. We conclude that, at least for file systems, code coverage metrics do not strongly correlate with test effectiveness, *i.e.*, the ability to detect bugs.

2.2 Keys for Bug Detection

Given the limited effectiveness of code coverage, we further investigated why covering code does not always reveal bugs, and identified key factors for detection. To this end, we analyzed each bug to determine the test cases (including syscalls and their arguments) needed to find it. We observed that many bugs can only be detected when specific syscalls and particular arguments are used. Executing calls with ineffective argument values may cover the code but fail to expose the bug. We refer to bugs that require specific argument values to trigger them as *input bugs*. Moreover, we found that many bugs occur in the exit or error paths of kernel functions, potentially affecting syscall return values and error codes [31, 54]; we refer to these as *output bugs*. Covering syscall return behavior is crucial for revealing them.



```
Bug fix in fs/ext4/xattr.c, v6.0-rc1
lsetxattr(size, ...)
  sys_lsetxattr(size, ...)
...
   vfs_setxattr(size, ...)
...
    ext4_xattr_set(value_len, ...)
...
     int ext4_xattr_ibody_set(inode, ...) {
-      if (EXT4_I(inode)->i_extra_isize == 0)
+      if (!EXT4_INODE_HAS_XATTR_SPACE(inode))
        return -ENOSPC;
```

Figure 1. An ext4 bug that qualifies as both an input and output bug. The bug was fixed by checking whether the inode has room to store additional xattrs in `ext4_xattr_ibody_set`.

Figure 1 shows an ext4 bug [78], fixed in Linux kernel version v6.0-rc1, which qualifies as both an input and an output bug. It is an input bug because it occurred only when `lsetxattr` used the maximum legal `size` argument, causing the minimum offset (`min_offs`) between two block groups to overflow. Although its lines, function, and branches are all covered by xfstests, the test suite failed to detect it. It is also an output bug because it occurs on a function's exit path and affects the behavior of an error code (*i.e.*, `ENOSPC`).

To determine a bug's classification as an input bug, output bug, both, or neither, we analyzed each bug in terms of the inputs required to trigger it and the outputs it can impact. Of the 70 bugs analyzed, we identified 50 as input bugs (71%), 41 as output bugs (59%), and 57 as either input or output bugs (81%). The prevalence of input and output bugs (or both) underscores the necessity of ensuring thorough coverage of inputs and outputs in file system testing. There are 13 non-input/output bugs (19%) that cannot be reliably triggered solely by inputs, nor do they occur on an exit path. For example, the bug [88] arises from a race condition; *i.e.*, not directly from how a syscall is invoked, but rather due to timing, thread interleaving, and mistaken dirtying of inodes during eviction.

Additionally, of the 37 bugs missed by xfstests despite covering their lines of code, 28 (76%) are input or output bugs. Among them, 24 (65%) are input bugs that require specific syscall argument values to be triggered. These argument values often involve corner cases, such as out-of-bound reads [8], less-tested inputs such as extended attributes [7, 37], and links [6], and boundary values that trigger overflow [78, 80] or are zero-length [81].

Code-coverage metrics typically do not take into account the diversity of input cases, with lightly tested inputs often following the same execution paths as well-tested ones [76]. Consequently, code coverage does not weight repeatedly executing the same code path with varying inputs [30]. Similarly, bugs in various output cases, such as error codes [42, 54], are not properly evaluated by such metrics.

Thus, we need to consider comprehensive coverage of input types, including syscalls and their arguments, as well as outputs such as return values and error codes. Given the lack of established metrics and tools for measuring input and output coverage in file system testing [40], we propose input and output coverage metrics and present IOCov as a means for evaluating them. In summary, code coverage alone is insufficient for testing, as many bugs rely on specific inputs and outputs that may be missed. Covering syscall inputs and outputs during testing helps address these limitations.

3 IOCov Metrics and Framework

We present new coverage metrics for file system testing: *input coverage* and *output coverage*, along with the IOCov framework for computing these metrics in testing tools. We begin with the input and output partitioning scheme we use for defining input and output coverage. We then formalize these coverage metrics and describe the architecture of IOCov, highlighting the filtering mechanism it uses to ensure accurate coverage computation.

3.1 Input and Output Coverage

Linux provides over 400 system calls, with many specifically related to file systems [5, 76]. Each system call can take multiple arguments, and both the arguments and the outputs can assume arbitrary values from large domains. Consequently, it is infeasible to evaluate whether all possible inputs and outputs are covered by testing. We observed, however, that file system calls exhibit structured input and output patterns, which can be partitioned into categories containing semantically similar cases. For example, the `open` syscall has a bitwise `flags` argument, where each flag represents a specific behavioral option. Enabling a particular flag triggers behavior tied to a distinct aspect of the syscall. For example, setting `O_CREAT` causes a file to be created if it does not already exist. Instead of analyzing all flag combinations, which would be exponentially complex, we treat each flag independently and check whether it appears in any test input. This reveals how well a testing tool covers the behaviors encoded by `open` flags. This partitioning strategy can also be applied to other inputs and outputs, but different input and output types require different methods.

For inputs, we classified syscall arguments into five categories: *identifier*, *pointer*, *bitmask*, *numeric*, and *categorical*. Identifiers include file and directory pathnames, as well as file descriptors that specify the object on which the syscall operates, such as `pathname` in `open` and `fd` in `write`. Pointer arguments refer to memory addresses that point to buffers or structures, such as `buf` in `write`. Bitmasks are arguments that can be logically `OR`-ed, such as `open`'s `flags` and `chmod`'s `mode`. Numeric arguments are scalars, often representing quantities such as the number of bytes in `write`'s `count` argument. Categorical arguments are discrete values chosen from a small set of options, such as `lseek`'s `whence`.

For input coverage, we exclude identifiers and pointers because their values correspond to specific operands or memory addresses and do not represent semantically distinct test cases. We partition numeric arguments using boundary-value analysis [18, 61, 65, 89], selecting powers of 2 as boundaries since they are commonly used in file systems [36]. Each partition is defined as the range between two adjacent boundaries. In the case of categorical arguments, each predefined option corresponds to a unique input partition.

We partition the outputs in a manner similar to categorical and numeric inputs. Most syscall outputs return either success or an error code. Accordingly, we divide the output space into success and failure, and further subdivide the failure cases by specific error codes. For syscalls that return a byte count on success (*e.g.*, `write`), we again partition outputs using powers of 2 as boundaries.

Partitioning the input/output space enables us to define coverage based on semantic groupings, eliminating the need to examine every possible value. As such, we define *input coverage* and *output coverage* as metrics that describe the extent to which a testing tool exercises the input and output partitions of file system calls. To formalize input coverage, consider a file system call S that has an argument A whose input space is partitioned into a set $\mathcal{I}$ of input classes. For a testing tool t, we define $C_{\text{in}}(t) \subseteq \mathcal{I}$ as the set of input partitions for A that are exercised by t. Then, the input coverage of argument A for syscall S is defined as:

$$\text{InputCoverage}_S^A(t) = \frac{|C_{\text{in}}(t)|}{|\mathcal{I}|}.$$

Similarly, consider a syscall S whose observable outputs can be partitioned into a set $\mathcal{O}$ of output classes. For a testing tool t, let $C_{\text{out}}(t) \subseteq \mathcal{O}$ denote the set of output partitions observed during test execution. Then, the output coverage of S is defined as:

$$\text{OutputCoverage}_S(t) = \frac{|C_{\text{out}}(t)|}{|\mathcal{O}|}.$$

Input and output coverage offer insights for improvement in both *completeness*, which measures whether all defined input and output partitions are exercised at least once (*i.e.*, coverage equals 1), and *balance*, which measures how evenly test executions are distributed across those partitions. Therefore, *coverage* quantifies how much of the partition space is exercised, while *completeness* ensures all partitions are covered and *balance* reflects how evenly they are tested. Ideally, a tool should cover as many input and output partitions as possible to ensure comprehensive coverage and achieve completeness. Moreover, given limited resources, it is important to avoid over-testing certain partitions and under-testing others to ensure balanced coverage.

The practical use of input and output coverage depends on the specific strategies and objectives of the testing process. Some tests tend to prioritize widely used partitions (*e.g.*, `O_RDONLY` for the `open` flag) because they are more likely to trigger common behaviors or bugs observed in real-world usage, while others prioritize less-used partitions (*e.g.*, `O_LARGEFILE`) to increase the likelihood of exposing rare or corner-case bugs. For output coverage, tests can define the expected output semantics and verify whether the observed results (*e.g.*, error codes) are correct. Exercising a wide range of output classes can uncover diverse test behaviors.

As such, the goal of input and output coverage is not to achieve perfectly balanced 100% coverage, but to help developers evaluate and improve testing by providing insights that code coverage alone cannot offer, such as which tests should be added and what error scenarios should be exercised.

3.2 IOCov Architecture

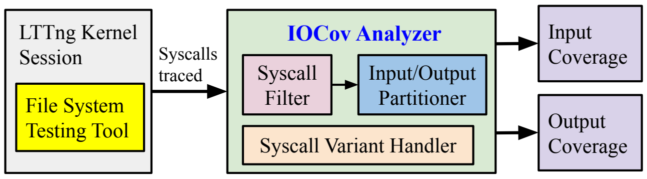


Figure 2. IOCov architecture and components.

With input and output coverage defined, the next step is to enable testing tools to compute these metrics accurately and efficiently. Doing so involves tracking the syscall inputs and outputs exercised by the tool. A concern is that although existing tracers can capture the syscalls exercised by the tool, not all captured syscalls originate from the test workload itself. Examples include `open` and `read` for loading libraries, or `write` for logging. These should not be included in coverage computation, as they do not exercise the tested file system. In particular, these calls are not directly triggered by the test input and therefore should not be included in input coverage; likewise, their outputs should not be considered part of output coverage.

We designed and implemented IOCov to address the limitations of existing tracers and to accurately compute input and output coverage. Figure 2 illustrates IOCov's architecture, its core components, and the workflow for computing coverage with a file system testing tool. We leverage LTTng [17] to trace the syscalls. Compared to other tracers such as `strace` [72] (which incurs high overhead), `ftrace` [43] (which may require manual processing for interpretation), and `bpftrace` [10] (which requires custom scripts and may face scalability challenges under high-frequency, multi-threaded workloads), LTTng offers low-overhead, out-of-band tracing with full syscall input/output capture and good scalability [1]. IOCov executes a given file system testing tool within an *LTTng Kernel Session*, allowing all syscalls invoked by the tool, including their inputs and outputs, to be recorded.

Table 1. Base syscalls and their variants supported by IOCov, along with the arguments used for input coverage. Each argument is annotated with its type: B (bitmask), N (numeric), and C (categorical).

Base Syscall	Variants	Supported Arguments
`open`	`openat`, `creat`, `openat2`	`flags (B)`, `mode (B)`
`read`	`pread64`	`count (N)`, `offset (N)`
`write`	`pwrite64`	`count (N)`, `offset (N)`
`lseek`	`llseek`	`offset (N)`, `whence (C)`
`truncate`	`ftruncate`	`length (N)`
`mkdir`	`mkdirat`	`mode (B)`
`chmod`	`fchmod`, `fchmodat`	`mode (B)`
`setxattr`	`lsetxattr`, `fsetxattr`	`size (N)`, `flags (B)`
`getxattr`	`lgetxattr`, `fgetxattr`	`size (N)`

The *IOCov Analyzer* then processes the syscalls traced by LTTng. It has three components: the *Syscall Filter*, the *Input/Output Partitioner*, and the *Syscall Variant Handler*. The *Syscall Filter*, described in detail in Section 3.3, analyzes raw traces and removes noisy or unrelated syscalls. Once the syscalls relevant for testing are identified, the *Input/Output Partitioner* collects their inputs from `syscall_entry` events and outputs from `syscall_exit` events, determines which partition each input and output value belongs to based on the method described in Section 3.1, and counts the occurrences of each input and output partition.

We observed that the testing tool can trigger file system calls that perform equivalent functions. For example, the `openat` syscall serves a similar purpose as `open` but allows one to specify a directory file descriptor for more flexible path resolution. We refer to the original or earliest form of a syscall, such as `open`, as a *base syscall*, and to the extended or modified versions derived from it as *syscall variants*. Since these variants share much of the same kernel implementation as their corresponding base syscalls [63, 76], the *Syscall Variant Handler* groups the base syscall and its variants together and merges their input and output spaces when computing coverage in the IOCov analyzer.

Table 1 lists the supported syscalls, along with their arguments and categories as defined in Section 3.1. IOCov supports 23 syscalls (nine base and 14 variants) for coverage computation, and also collects additional syscalls (*e.g.*, `close`, `chdir`) to help identify file descriptors and pathnames associated with the test workload. After processing by the

IOCov Analyzer, a report of the testing tool's input and output coverage is produced using a nested JSON key-value format, capturing the occurrence count of each input and output partition for every syscall supported by IOCov.

3.3 IOCov Filtering Mechanism

A key procedure in IOCov that differentiates it from other syscall tracers is its filtering mechanism, which retains only calls relevant to the test workload. IOCov exploits the fact that most testing tools use a dedicated mount point to exercise the tested file system. This approach provides an isolated and controlled testing environment while preventing any impact on existing file systems. For example, xfstests uses `/mnt/test` as the default mount point for executing tests, and CrashMonkey uses `/mnt/snapshot`. File system calls specify the target object using either file descriptors or pathnames. By checking whether the accessed object resides under the test mount point, we can determine whether a syscall belongs to the testing workload or is unrelated noise.

Algorithm 1 illustrates how IOCov filters syscalls. The LTTng trace file is generated during execution to log each `syscall_entry` event (recording inputs) and each `syscall_exit` event (recording outputs), along with additional information such as a strictly increasing timestamp. IOCov parses each line to obtain the type (entry or exit), syscall name, and arguments (including file descriptors and path names) or return values. Using this information, we determine whether each line is related to testing by examining its file descriptor (`fd`) or pathname (`path`). The `IsTesting` function in Algorithm 1 constructs the full pathname from the `path` and the current working directory (`cwd`), which is necessary for handling syscalls such as `openat` that may interpret paths relative to a directory `fd` (*e.g.*, `AT_FDCWD`). The function then checks whether the resulting path belongs to the test mount point. If the syscall is `open` or one of its variants, and both `IsTesting` returns true and the syscall completed successfully, the returned file descriptor is added to `fd_set` to represent file descriptors associated with testing.

Additionally, we handle `close` and `close_range` syscalls to remove expired testing-related file descriptors. Upon a successful `close`, the closed file descriptor is removed from `fd_set`. We also process `chdir` and `fchdir` syscalls, updating `cwd` to reflect changes to the current working directory. In this way, when a file system call is detected (*i.e.*, `IsFSCall` returns true), and its file descriptor is in `fd_set` or its pathname passes the testing check, the syscall is considered testing-related, and its inputs and outputs are added to `filtered_syscalls` for coverage computation.

Algorithm 1: Syscall filtering in IOCov to retain only those related to testing. `opened_fd` tracks descriptors opened for testing, and `closed_fd` tracks those pending closure; once a `close` syscall succeeds, the descriptor is no longer considered valid or open.

```
Input: trace_file: trace output of the testing tool
Output: filtered_syscalls: test-related traces
1  Initialize filtered_syscalls ← [ ] ;
2  Initialize fd_set ← ∅ ;          // A set of file descriptors (fd) used for testing
3  Initialize cwd ← current working directory
4  foreach line in trace_file do
5      Parse line to get event_type, syscall, fd/path;
6      if fd ∈ fd_set or IsTesting(cwd, path) then
7          if event_type = syscall_entry then
8              if IsOpen(syscall) then
9                  Add fd to opened_fd;
10             else if IsClose(syscall) then
11                 Add fd to closed_fd;
12             else if IsFSCall(syscall) then
13                 Add syscall inputs to filtered_syscalls;
14         else if event_type = syscall_exit then
15             if IsOpen(syscall) and return ≠ −1 then
16                 Add opened_fd to fd_set;
17             else if IsClose(syscall) and return = 0 then
18                 Remove closed_fd from fd_set;
19             else if IsChdir(syscall) and return = 0 then
20                 Update cwd;
21             else if IsFSCall(syscall) then
22                 Add syscall outputs to filtered_syscalls;
```

4 CM-IOCov: IOCov for CrashMonkey

This section demonstrates how IOCov can be used to improve CrashMonkey, a file system testing tool for crash consistency. We created CM-IOCov as an improved version of CrashMonkey with greater input coverage, and present its design and architecture in this section.

4.1 CM-IOCov Architecture

CrashMonkey [58, 59], which tests file system correctness under crash scenarios, does not actually crash the file system. Instead, it simulates crashes by recording I/O and replaying it up to a persistence point (*e.g.*, after an `fsync()` call). It generates test workloads as short sequences of syscalls, each followed by an explicit persistence operation such as an `fsync` or a global `sync`. It then compares the file system state after a simulated crash to a corresponding oracle, which represents the expected state after a safe unmount, and treats any mismatch as a crash consistency bug.

Table 2. Comparison of syscall inputs generated for testing in CrashMonkey and CM-IOCov, showing how CM-IOCov improves input coverage over CrashMonkey.

Syscall	File System Operation	Input	CrashMonkey	CM-IOCov
`open()`	Create writable file	mode	`0777`: full permissions	Multiple read/write modes
		flags	`O_CREAT | O_RDWR`: read-write	Various flags and combinations
`mkdir()`	Create writable directory	mode	`0777`: full permissions	Multiple directory modes
`write()`/ `fallocate()`	Append to file end	size (in bytes)	Write: fixed size 32768	Aligned writes with varied sizes
	Overwrite from file start		Write: offset 0, size 5000	Unaligned writes with varied sizes
	Overwrite near file end		Write: size 5000 near file end	Unaligned writes with varied sizes near file end
	Overwrite and extend file		Offset before EOF, size 5000, extends by 3000	Multiple sizes and offsets to overlap and extend
`truncate()`	Block-aligned truncate	length (in bytes)	Truncate to length 0	Multiple aligned sizes
	Unaligned truncate		Truncate to length 2500	Multiple unaligned sizes

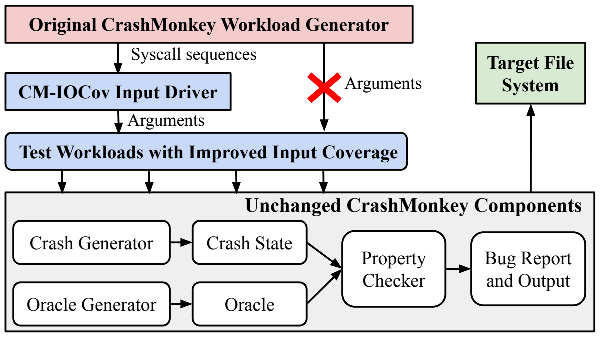


Figure 3. CM-IOCov architecture: improves CrashMonkey's test workload generation by introducing an input driver that provides syscall inputs with higher input coverage.

CM-IOCov improves CrashMonkey's test workload generation while reusing its crash simulation, oracle generation, and state comparison components. By only improving input coverage, we isolate its impact from other potential improvements to clearly observe its effect. Figure 3 illustrates the CM-IOCov architecture. The original CrashMonkey workload generator first builds syscall sequences and then fills in the arguments to satisfy operation dependencies (*e.g.*, creating a file before accessing it). CM-IOCov uses the same syscall sequences and dependency resolution strategy for file and directory objects as CrashMonkey, but employs the *CM-IOCov Input Driver* to generate argument values that offer better input coverage for syscalls supported by IOCov.

CrashMonkey's argument selection for syscalls relies solely on manually specified fixed values. For example, when creating a file using `open()`, it always uses mode `0777`, which grants read, write, and execute permissions to everyone, without considering the diversity of permission settings that may affect file system behavior. Thus, by using the *CM-IOCov Input Driver* instead of CrashMonkey's fixed argument-selection strategy, CM-IOCov generates test workloads with better input coverage, while reusing the other CrashMonkey components to simulate crashes, verify post-crash states against the oracle, and detect bugs.

4.2 CM-IOCov Input Driver

CM-IOCov's key component is its *Input Driver*, designed to generate more diverse syscall arguments than CrashMonkey, thereby achieving improved input coverage and finding bugs that CrashMonkey misses. Table 2 provides a list of syscalls and their inputs where CM-IOCov improves upon CrashMonkey. In particular, for `open()`, CrashMonkey uses fixed values for the `mode` and `flags` arguments, `0777` and `O_CREAT | O_RDWR` respectively, which always creates a file with full permissions and read-write access for all users, but fails to explore other file creation scenarios. CM-IOCov creates files using a broader combination of modes and flags than CrashMonkey, including various read, write, and execute permission settings for non-owner users and groups, as well as additional file creation flags such as `O_TRUNC`, which truncates a file to zero length if it already exists, and `O_APPEND`, which ensures that all writes are appended to the end of the file. For numeric arguments such as the size in `write`/`fallocate` and the length in `truncate`, CrashMonkey uses a fixed value across all cases. For example, when appending to a file, it always writes 32,768 bytes, regardless of the file system type or underlying disk size.

By contrast, CM-IOCov incorporates multiple input generators tailored to each scenario involving byte arguments, including power-of-two values for aligned cases and $2^n \pm 1$ values to capture unaligned cases and edge conditions near alignment boundaries. This enables exploration of a wider range of numeric byte values, achieving better input coverage than CrashMonkey in cases such as appending to a file, overwriting, or truncating a file. For test cases involving file extension via `write`, CM-IOCov uses two input generators: one for the offset and one for the write size. This ensures coverage of both overlapping writes and writes that extend the file. CM-IOCov also takes the file system's disk size into account when generating byte arguments so that file writes stay within the available space, thereby avoiding `ENOSPC` failures. CM-IOCov also supports additional file system operations present in CrashMonkey that are not shown in Table 2, such as direct-IO write and `mmap` write, for which the input driver can improve syscall inputs.

For each argument supported by IOCov, CM-IOCov constructs a value pool containing relevant inputs, such as those produced by the byte argument generators mentioned above, to achieve high input coverage for various test scenarios. Once the syscall sequence is determined, CM-IOCov randomly selects a value from the corresponding argument pool for each argument. The development effort for writing new tests in CM-IOCov is minimal, as users can either use the CM-IOCov driver directly or create their own tests based on input and output partitions. Compared to CrashMonkey, CM-IOCov generates the same syscall sequences but varies argument values to achieve broader coverage.

Unlike semantic file system testing, which also examines error cases [66], crash consistency testing requires syscalls to execute successfully in order to verify crash-time properties [15], such as whether a successfully created file persists after a crash. Therefore, CM-IOCov excludes inputs that could cause file system operations to fail, such as invalid flags that prevent file creation or write sizes that exceed the available disk space. Due to the vast number of syscall sequences [58], randomized input selection from value pools enables coverage of diverse input spaces and improves overall input coverage.

5 Implementation

IOCov is implemented in three main components: (1) a *parser* that analyzes and extracts coverage information from LTTng trace logs; (2) a *coverage visualizer* that plots and displays input and output coverage to aid developer analysis; and (3) a set of *scripts* that run various file system testing tools within an LTTng session and perform component integration. The implementation of IOCov comprises 3,045 lines of code, of which 2,990 are written in Python and 55 in Bash. The parser for log processing and syscall filtering contains 1,029 lines of Python code. The visualizer, implemented in 410 lines of Python code, displays input and output coverage to help developers identify uncovered areas and improve testing accordingly. The remaining lines of code are supporting scripts and essential utilities used to automate and integrate the various components.

Implementing CM-IOCov required us to modify the original CrashMonkey and add an input driver. We modified 381 lines of CrashMonkey, including its workload generation unit and shell scripts, and implemented the CM-IOCov input driver in 211 lines of Python code. The CM-IOCov input driver automates input generation for multiple syscalls and can potentially support diverse inputs for other analysis tools as well. Additionally, the original CrashMonkey includes two kernel modules: one for I/O recording and another for taking block device snapshots. It supports Linux kernel versions only up to 5.6.

To evaluate CM-IOCov on Linux kernel version 6.12, the latest version at the start of our kernel migration, and uncover realistic bugs, we updated the system to be compatible with that kernel. The modifications consist of 573 insertions and 944 deletions in C source files, headers, and the Makefile related to the CrashMonkey kernel modules. To support Linux kernel 6.12, we updated kernel APIs and macros for compatibility, improved block I/O dispatch, simplified disk and queue handling, and removed obsolete code.

Implementation Challenges. While the input and output coverage metrics provided by IOCov help developers evaluate and enhance testing, and CM-IOCov improves existing crash consistency testing, several challenges still remain. First, the coverage metrics depend on how we partition the input and output space, which may leave certain cases uncovered. For example, we compute coverage of the `open` syscall by considering all of its flags individually. We do not, however, account for combinations of flags, which is also important for generating meaningful test cases. Computing coverage over all flag combinations is infeasible, as the flag field in `open` can represent up to 2^{23} possible values (many illegal) due to bitwise combinations. A second challenge is that the capability of certain file system testing tools to improve input and output coverage is constrained. For example, CrashMonkey mandates that syscalls succeed, which prevents it from exploring outputs in the form of error codes.

Therefore, the effectiveness of testing improvements depends on the nature of the testing goal. Output coverage is important for semantic testing [56, 66], but it is not essential for crash consistency testing [41, 58, 69]. Additionally, different testing tools may focus on specific input or output partitions, such as writing many small files or a single large file [57]; thus, aligning input/output coverage with test goals still requires domain knowledge and manual effort. Nevertheless, input and output coverage, along with IOCov, provide an effective methodology for developers to use to evaluate and enhance testing more easily than traditional code coverage. We demonstrated the utility of our approach by using it to develop CM-IOCov and thereby improving existing crash consistency testing.

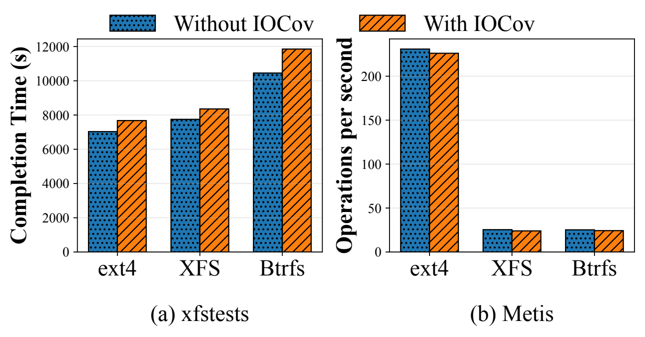


Figure 4. Performance overhead of IOCov on xfstests and Metis, measured using completion time and operation rate, respectively.

6 Evaluation

In this section, we address the following questions: **(1)** What are the overhead and performance impacts of applying IOCov to file system testing tools (§6.1)? **(2)** How accurately does IOCov compute input and output coverage in file system testing (§6.2)? **(3)** How can IOCov be utilized to assess the coverage of existing testing tools and provide developers with insights for improvement (§6.3)? **(4)** Can CM-IOCov improve bug detection and discover bugs that the original CrashMonkey fails to detect (§6.4)?

Experimental setup. We conducted all IOCov experiments on two virtual machines (VMs), each equipped with 8 CPU cores and 128GB of RAM. Both VMs ran Ubuntu 22.04 with Linux kernel version 5.19.6. We ran all CM-IOCov and CrashMonkey experiments on two additional VMs configured identically to the IOCov machines, except that one used Linux kernel version 5.6 (the latest version supported by the original CrashMonkey) and the other used version 6.12, which is the kernel we migrated the system onto. Each VM was equipped with a 1TB disk partition used to store all generated executables and log files. To evaluate IOCov, we employed four file system testing tools: xfstests [68], Metis [46], Syzkaller [27], and CrashMonkey [58]. These tools were selected as representatives of different testing techniques: regression testing, model checking, fuzzing, and automatic test generation, respectively. As the tools vary in design and capabilities, we selected appropriate tools for each evaluation task, while measuring input and output coverage across all of them.

6.1 IOCov Overhead

Applying IOCov to testing tools to compute input/output coverage introduces additional overhead that may slow testing. First, IOCov relies on LTTng to trace system call inputs and outputs, introducing additional CPU and I/O overhead for capturing events and writing trace logs. Second, computing the final input and output coverage requires analyzing the logs to extract the relevant coverage information.

To evaluate overhead, we applied IOCov to two testing tools: xfstests and Metis, which respectively rely on manually crafted test cases and automated testing (state exploration). The xfstests suite includes two test-case categories: *generic tests*, applicable to all file systems, and *specialized tests*, designed for specific file systems and their unique features. Metis generates diverse inputs to systematically explore file system states and in the process check if the system behavior is as expected. The tests in xfstests are fixed, so we measure overhead by comparing the completion time with and without IOCov. Metis, in contrast, explores states dynamically at runtime and can run for extended periods. Therefore, overhead is measured as the difference in Metis's speed (operations per second) when run with and without IOCov.

Figure 4 compares the performance of xfstests and Metis with and without IOCov. Using three file systems (ext4 [55], XFS [70], and Btrfs [67]), we executed all of xfstests' generic tests and ran Metis for one hour. With IOCov, the overhead from LTTng increased xfstests' completion time by 7.8–13.4%. For Metis, more operations per second indicates better performance; with IOCov, the operation rate dropped by 2.1–5.3%. Metis runs faster on ext4 than on XFS and Btrfs because it uses the minimum allowed device size: 2 MiB for ext4 compared to 16 MiB for the others. Metis takes longer to save and process states for larger devices. For post-tracing analysis, IOCov took 22.5 minutes to extract coverage information for xfstests applied to ext4 and 17.4 minutes for Metis.

The extracted input and output coverage data for all supported syscalls, formatted as nested JSON, was only 2.8 MB, whereas the raw LTTng trace from xfstests was 41 GB, indicating that IOCov efficiently extracts coverage information from large traces. In summary, IOCov introduces a small and acceptable compute and storage overhead when extracting input and output coverage.

6.2 IOCov Accuracy

It is important that IOCov accurately filter syscalls related to file system testing for coverage computation, a property we refer to as its *accuracy*. To evaluate accuracy, we compared its reported coverage with the verified inputs and outputs exercised during testing, which served as the ground truth. Many file system testing tools do not log the syscalls they execute for testing (*e.g.*, xfstests), or retain only incomplete information (*e.g.*, CrashMonkey). As such, we applied Metis to this task, as it records file system calls along with their inputs and outputs, capturing only those relevant to testing.

This accuracy experiment compared the coverage reported by IOCov for Metis against the inputs and outputs recorded in Metis logs. We ran Metis for one hour on ext4 while using LTTng to trace syscalls. We then computed input/output coverage using IOCov and measured the expected coverage

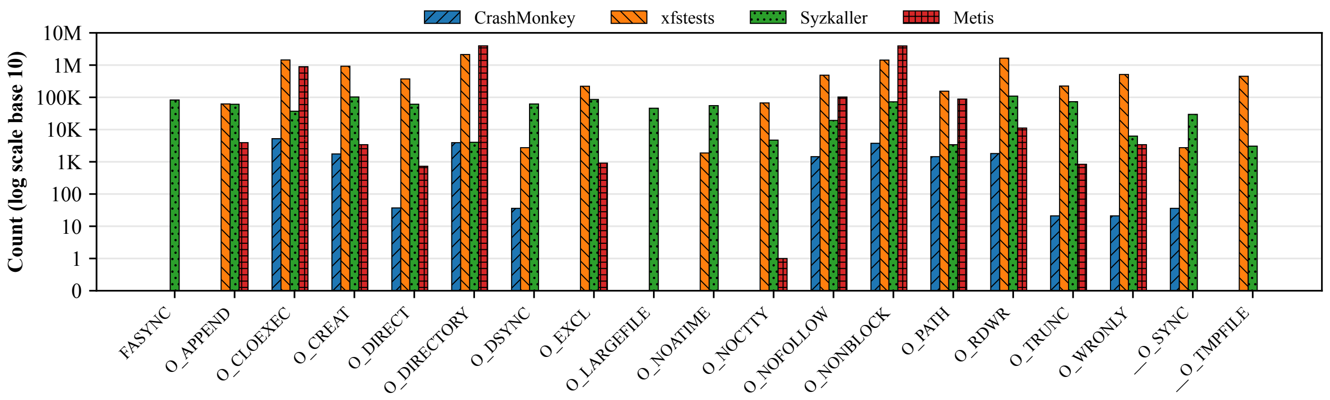


Figure 5. Log$_{10}$-scaled input coverage counts (y-axis) for each open flag (x-axis), measured across CrashMonkey, xfstests, Syzkaller, and Metis.

based on Metis logs. For the comparison, we counted the file system calls captured by IOCov (computed coverage) and those recorded in the Metis logs (expected coverage). We found that both recorded similar counts for most file system syscalls. IOCov recorded 39,641 calls to `write`, 523,152 to `truncate`, 29,280 to `mkdir`, and 54,486 to `chmod` while Metis reported 41,935 calls, 523,047 calls, 29,261 calls, and 54,475 calls, respectively. The highest error rate among the syscalls was for `write` at 5.47%, which is acceptable for understanding the testing tool's input and output coverage. The mismatch comes from slight LTTng instrumentation limitations or timing discrepancies, which may cause certain syscalls to be missed or logged inaccurately compared to ground-truth execution.

Additionally, we examined partitions of syscall inputs and outputs in detail to further assess accuracy. IOCov relies on the `open` syscall to filter other syscalls and compute coverage. To compare `open` flags between IOCov and the Metis logs, we examined the coverage of each of the 21 flags individually. We found that 13 flags had identical coverage in both; for example, `O_WRONLY` appeared 3,397 times, `O_EXCL` 913 times, and `O_DIRECT` 729 times. The remaining flags showed slight variations; for example, `O_RDWR` appeared 11,160 times in Metis logs and 11,166 times in IOCov, `O_CREAT` 3,397 vs. 3,398, and `O_APPEND` 3,929 vs. 3,931. Among the flags with discrepancies, the largest difference is with `O_DIRECTORY`, where Metis reported 1,693,264 occurrences and IOCov computed 1,711,525, resulting in a 1.08% error rate. The `O_DIRECTORY` flag, used for opening directories, appeared far more frequently than the others because Metis traverses the entire file system after each operation to compute a hash of the resulting state. Overall, IOCov accurately retains test-related syscalls and thereby computes input and output coverage for file system testing tools with high accuracy.

6.3 IOCov Use Cases in Testing

We used IOCov to measure input and output coverage for different types of testing tools in order to evaluate them and deliver insights on how they can be improved. Our evaluation covers four tools: CrashMonkey [58], xfstests [68], Syzkaller [27], and Metis [46]. These tools vary in their characteristics and runtime behavior. CrashMonkey generates grouped workloads based on test length; xfstests runs a fixed set of workloads, with total duration depending on the selected test groups; Syzkaller and Metis continuously generate workloads and can run for extended periods. To ensure fairness, we selected workloads for CrashMonkey (including `seq-1` and other default workloads) and xfstests (the `quick` group) that complete within one hour. We then ran Syzkaller and Metis for the same duration. Each tool was evaluated using the same underlying file system, ext4. Specifically, Metis allows flexible configuration of syscall input distributions; however, we used its default settings without additional customization. Although Syzkaller is not a dedicated file system fuzzer, it can trigger file system bugs through relevant syscalls, so we restricted it to generate only file system-related calls. While IOCov is capable of generating comprehensive coverage data for all supported syscall inputs and outputs, due to space constraints we report and analyze only a representative subset.

Figure 5 shows input coverage for `open` flags across these four tools. The x-axis shows individual flags, each representing a single bit, and the y-axis (log$_{10}$ scale) indicates how frequently each flag was exercised by the testing tool. Input coverage for `open` flags is measured over all input partitions, *i.e.*, the individual flags that compose the bitmask. Among the four tools, Syzkaller achieves the most thorough input coverage for `open` flags—exercising all flags and being the only tool that covers both `FASYNC` and `O_LARGEFILE`. Although Metis and xfstests cover many commonly tested flags, they miss certain ones that are also worth testing. For

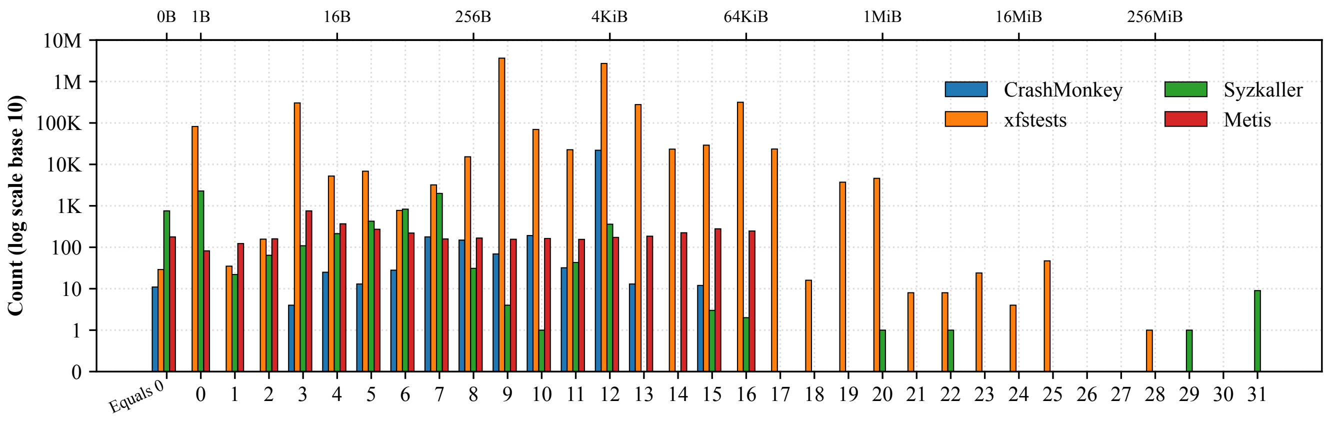


Figure 6. Log_{10}-scaled input coverage counts (y-axis) for `write` sizes (in bytes), measured across CrashMonkey, xfstests, Syzkaller, and metis. The x-axis shows powers of 2 for write sizes, with a special entry labeled "Equals 0" for writes of size zero.

example, `O_LARGEFILE`, which enables support for large files, may still expose bugs [79] in modern systems. CrashMonkey covers the fewest flags and even misses common ones such as `O_APPEND`, indicating a need for improved input coverage (which led to the development of CM-IOCov). Furthermore, our results show that some flags are exercised millions of times, while others are never tested, revealing a significant imbalance in test coverage. Such information can inform the reallocation of test efforts to under-tested cases.

Figure 6 shows the input coverage for the `write` size argument (count in bytes), partitioned by boundary values based on powers of 2. The x-axis represents the $\log_2$ of the `write` size, while the $x2$-axis shows the corresponding actual size. For example, $x = 8$ represents all sizes from 2^8 to $2^9 - 1$ (*i.e.*, 256–511 bytes), with the corresponding $x2$-axis value shown as 256 B. The y-axis ($\log_{10}$) shows how many times each x-axis bucket was tested by a given tool. As shown in Figure 6, all four tools prioritize testing small sizes (less than 4 KiB) and lack coverage for larger sizes. While Syzkaller and xfstests covered a broader range of `write` size partitions than Metis and CrashMonkey, all four tools completely missed some partitions. Similarly, all tools showed uneven testing of `write` sizes. For instance, xfstests exercised some size partitions millions of times, while others were not tested at all. This coverage information provides developers with direct insights into how to improve test cases and address untested scenarios.

We omit output coverage results due to space limitations, but our observations showed that all testing tools prioritized successful syscall returns and missed many error scenarios, such as `ETXTBSY` related to concurrency and `EOVERFLOW` for oversized values in `open`.

6.4 CM-IOCov Bug Finding

IOCov's ultimate objective is to enhance existing test tools to uncover file system bugs that the originals miss. We developed CM-IOCov to enhance CrashMonkey's crash consistency testing by leveraging IOCov's coverage reports, and evaluated whether it finds more bugs than the original CrashMonkey on Linux kernel versions 5.6 and 6.12. Linux 5.6 is the latest kernel that the unmodified CrashMonkey supports, but bugs in this version may already be fixed in newer kernels. We ported both CrashMonkey and CM-IOCov to Linux 6.12 by updating their code and kernel API usage, allowing them to run on the newer kernel and discover more recent bugs. We evaluated the Btrfs file system, which is the main file system targeted by CrashMonkey [58], on both kernels.

To validate CM-IOCov's effectiveness, we generated the same set of test workloads for both CM-IOCov and CrashMonkey. Each workload consists of a short sequence of syscalls, followed by crash simulation and a checker to verify file system correctness after crashes. If a test workload fails, meaning the checker detects an incorrect state after a crash, it indicates a potential crash consistency bug. However, the number of failed workloads does not reflect the number of unique bugs, because one bug can cause multiple failures.

In the kernel 5.6 experiment, both CM-IOCov and CrashMonkey executed 426,238 test workloads. CM-IOCov found 3,200 failures compared to CrashMonkey's 2,831, showing CM-IOCov's improved bug-finding capability. Both versions detected 2,800 common failures. In addition, CM-IOCov uncovered 400 failures missed by CrashMonkey, while CrashMonkey found only 31 that CM-IOCov did not. The results demonstrate that, with improved input coverage in CM-IOCov compared to CrashMonkey and all other factors unchanged, CM-IOCov achieves better bug detection, as evidenced by the higher number of failures identified.

Table 3. Crash consistency bugs in Btrfs, their consequences, and the triggering call sequences. These bugs were detected by CM-IOCov but missed by CrashMonkey on Linux 6.12. Underlines indicate inputs improved by CM-IOCov over CrashMonkey.

No.	Bug Consequence	System Call Sequence
1	Allocated blocks lost after `fsync`	<u>`open`</u>, <u>`write`</u>, <u>`falloc`</u>
2	File content did not match after `fsync`	<u>`open`</u>, <u>`write`</u>, <u>`mmapwrite`</u>
3	Data block missing after `rename`	<u>`open`</u>, <u>`write`</u>, <u>`falloc`</u>, `rename`
4	Rename not persisted by `fsync`	<u>`opendir`</u>, `close`, `rename`, <u>`mkdir`</u>
5	Incorrect number of file hard links after `fsync`	<u>`mkdir`</u>, <u>`open`</u>, `link`, `rename`

For the 6.12 kernel, we ran 391,134 test workloads using both CM-IOCov and CrashMonkey. During that experiment, CM-IOCov reported 390 failures, compared to 224 found by CrashMonkey. CM-IOCov found 323 failures missed by CrashMonkey, while CrashMonkey found only 157 missed by CM-IOCov, again demonstrating CM-IOCov's superior bug detection ability. Table 3 presents five representative Btrfs bugs identified by CM-IOCov but missed by CrashMonkey, along with their consequences and the syscalls that triggered them. Across the bugs listed in Table 3, as well as the failures uniquely identified by IOCov, most of the involved syscalls benefited from CM-IOCov's improved input generation; these syscalls are underlined in the table. This highlights the importance of input coverage for bug detection. The crash-consistency bugs found by CM-IOCov have serious consequences, such as allocated blocks being lost despite explicit persistence via `fsync()`, file content or hard link counts not being persisted, and files missing after a crash. Since these bugs were found on Linux kernel 6.12, they are likely to be real and still present. We are actively investigating them and plan to report them to Btrfs developers with detailed diagnostic information. While CrashMonkey did detect some failures that CM-IOCov missed under the same workload count, its inputs are a subset of CM-IOCov's. Therefore, with enough workloads, we believe CM-IOCov can also reveal those failures.

7 Related Work

This section discusses related work on test coverage metrics, code coverage effectiveness, and file system testing.

Test Coverage Metrics. Coverage metrics have long been a cornerstone of software testing, providing a quantitative method to evaluate the thoroughness of test suites [47, 53, 91]. There are general coverage metrics for most software and specialized ones for specific test targets [64]. As the most widely used general metric, code coverage includes subtypes such as line, statement, function, and branch coverage, categorized by the granularity of the code measured [34]. In file system testing, however, developers primarily conduct tests by issuing syscalls to file systems in kernel space. Due to the complex path between user-space test suites and kernel-space file system implementations, it is unclear which syscalls to issue to cover specific code [4, 23].

Specialized coverage metrics [84] differ across testing techniques because each technique targets different bug types, exercises different system features, and uses distinct methods to generate and run test cases. For example, file system model checking [46, 85] aims to achieve state coverage by exploring as many file system states as possible. Similarly, testing crash consistency requires one to cover diverse crash scenarios [2, 9, 58, 62]. Current approaches to file system testing primarily rely on generic coverage metrics, such as code coverage, without specifically evaluating their effectiveness for this domain [38]. Moreover, no formal coverage metrics have been defined explicitly for file system testing.

Although input and output coverage metrics exist [3, 40, 77], they are designed for different targets (*e.g.*, quantum programs, network protocols) and are not suited for file system testing, which relies on syscall inputs and outputs.

Effectiveness of Code Coverage. Assessing the effectiveness of code coverage is an active area, but the findings remain inconclusive and lack consensus. We define *effectiveness* as the ability to detect faults or bugs in the test target [47]. Some studies [25, 26, 39] suggest that the effectiveness of code coverage is inconsistent and depends on the specific target. For example, Kochhar *et al.* [39] found a strong correlation between code coverage and test effectiveness for the Mozilla Rhino JavaScript engine, whereas the correlation was moderate for Apache HTTPClient. Some studies [21, 22, 28, 30] show that the correlation is contingent upon specific subclass metrics of code coverage. According to Gopinath *et al.* [28], statement coverage is the most effective metric for detecting faults, outperforming other code coverage metrics. Hemmati *et al.* [30] observed, however, that statement coverage is significantly weaker than other coverage metrics such as branch coverage.

Moreover, prior work [11, 12, 33, 60] indicates that code coverage has a limited correlation with test effectiveness, and thus new, complementary coverage criteria are needed [71]. Specifically, Inozemtseva *et al.* [33] analyzed 31,000 test suites for five systems and found a low-to-moderate correlation between code coverage and effectiveness. Nevertheless, there is no existing research examining this correlation for complex low-level systems, such as in-kernel file systems. This work investigates the correlation for file system testing through the bug study presented in Section 2.

File System Testing. File system testing can be static or dynamic, depending on whether it involves actively exercising the file system [56]. This paper focuses on dynamic testing, which generally has three steps: (1) generating test cases in the form of syscalls, (2) initializing the file system under test and executing the tests, and (3) validating file system properties post-execution. Here, we consider four representative methods: regression testing, model checking, fuzzing, and automatic test generation, and explain how they achieve coverage.

Regression testing (*e.g.*, xfstests [68] and LTP [57]) is a collection of tests manually crafted to ensure that updates or new features do not introduce bugs or failures. It often aims to achieve functionality coverage by testing as many file system features as possible to verify the correctness of each. Given, however, the continuous evolution of file system features and the handcrafted nature of regression testing, it is difficult to ensure complete functionality coverage or provide a measure to assess the completeness of the testing [4].

Model checking [46, 73, 85, 86] is a formal verification technique used to find bugs by automatically and systematically exploring a file system's state space. During exploration, model checking verifies whether each file system state adheres to a specification. State coverage, however, is not a practical metric for file system testing for two reasons: (1) due to the complexity of file systems, the number of possible states grows exponentially with the number of system components, a problem known as *state explosion* [16], and (2) state coverage does not provide clear guidance to developers on how to improve their tests. As states are often difficult to predict and accurately represent, developers may not know which specific test cases or scenarios need improvement [46].

Fuzzing [82, 83, 83] uses code coverage as guidance and employs heuristics to prioritize test inputs that increase coverage. However, it lacks guarantees for achieving comprehensive coverage or accessing hard-to-reach code paths.

Automatic test generation [14, 58] creates rule-based syscall workloads to test file system functionality and reliability, typically covering various combinations of syscalls. For example, CrashMonkey [58] exhaustively permutes syscalls within a defined bound to construct operation sequences for testing file system crash consistency. However, focusing solely on syscall combinations is inadequate, as each syscall can have different argument values, leading to different test cases for the same syscall [24].

To our knowledge, no prior work exists on designing coverage metrics for file system testing by studying real bugs, nor on enhancing testing using these metrics.

8 Conclusion and Future Work

In this paper, we first analyzed known file system bugs to reveal the limitations of traditional code-coverage metrics. We then presented two new metrics, input coverage and output coverage, for evaluating and improving file system testing. We created the IOCov framework to measure these metrics and developed CM-IOCov, an enhanced version of CrashMonkey with higher input coverage, enabling more effective detection of crash consistency bugs. Our evaluation shows that, with low overhead, IOCov accurately measures input and output coverage for file system testing tools, and identifies untested and unbalanced test cases to guide tool improvement. Our results with CM-IOCov show that improving input coverage can substantially enhance file system testing with modest effort, such as supplying an input driver with broader coverage, and yielding significant gains in test effectiveness and bug discovery. CM-IOCov discovered Btrfs bugs in recent Linux kernels that the unmodified version, CrashMonkey, failed to find.

Future Work. We aim to extend IOCov to support input and output coverage for additional file system calls, including `ioctl` and `mount`. We also seek to investigate which input and output partitions are most effective at revealing bugs. We believe that IOCov techniques can be extended to other system software, including databases, distributed systems, and operating system components. For example, measuring the coverage of SQL queries and their results can help improve database testing.

Acknowledgments

We thank the anonymous SYSTOR reviewers and our shepherd, Animesh Trivedi, for their valuable feedback. We also thank Gautam Ahuja, Rohan Bansal, and Tanya Raj for their contributions. This work was made possible in part thanks to Dell-EMC, NetApp, Facebook, and IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-1900589, CNS-1900706, CCF-1918225, CNS-1951880, CNS-2106263, CNS-2524208, CNS-2106434, CNS-2214980, CPS-1446832, ITE-2040599, and ITE-2134840.

References

[1] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. 2020. Re-Animator: Versatile High-Fidelity Storage-System Tracing and Replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR '20)*. ACM, Haifa, Israel, 61–74.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Savannah, GA, USA, 151–167.

[3] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs. In *Proceedings of the 14th IEEE Conference on*

Software Testing, Verification and Validation (ICST). IEEE, Porto de Galinhas, Brazil, 13–23.

[4] Naohiro Aota and Kenji Kono. 2019. File Systems are Hard to Test — Learning from Xfstests. *IEICE Transactions on Information and Systems* 102, 2 (2019), 269–279.

[5] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. 2018. Analyzing a decade of Linux system calls. *Empirical Software Engineering* 23 (2018), 1519–1551.

[6] Eric Biggers and Theodore Ts'o. 2022. Ext4: disable fast-commit of encrypted dir operations. *https://github.com/torvalds/linux/commit/0fbcb5251fc81b58969b272c4fb7374a7b922e3e*.

[7] Ye Bin and Theodore Ts'o. 2022. Ext4: fix inode leak in ext4_xattr_inode_create() on an error path. *https://github.com/torvalds/linux/commit/e4db04f7d3dbbe16680e0ded27ea2a65b10f766a*.

[8] Ye Bin and Theodore Ts'o. 2022. Ext4: Fix Potential Out of Bound Read in ext4_fc_replay_scan(). *https://github.com/torvalds/linux/commit/1b45cc5c7b920fd8bf72e5a888ec7abeadf41e09*.

[9] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Atlanta, GA, 83–98.

[10] Bpftrace Developers. 2025. bpftrace: High-level tracing language for Linux. *https://github.com/iovisor/bpftrace*.

[11] Lionel Briand and Dietmar Pfahl. 1999. Using Simulation for Assessing the Real Impact of Test Coverage on Defect Coverage. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society Press, Boca Raton, FL, USA, 148–157.

[12] Xia Cai and Michael R. Lyu. 2005. The Effect of Code Coverage on Fault Detection under Different Testing Profiles. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–7.

[13] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[14] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Testing File System Implementations on Layered Models. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, Seoul, South Korea, 1483–1495.

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

[16] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. 2018. *Model Checking, 2nd Edition*. MIT Press, Cambridge, MA.

[17] Mathieu Desnoyers and Michel R Dagenais. 2006. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Ottawa Linux Symposium (OLS)*, Vol. 2006. Citeseer, Ottawa, Canada, 209–224.

[18] Felix Dobslaw, Robert Feldt, and Francisco de Oliveira Neto. 2022. Automated Black-Box Boundary Value Detection. *arXiv preprint arXiv:2207.09065* abs/2207.09065 (2022), 27 pages.

[19] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Guiding Simulations of Multi-Tier Storage Caches Using Knee Detection. In *31st Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '23)*. IEEE Computer Society, Stony Brook, NY, 1–8. doi:*10.1109/MASCOTS59514.2023.10387545*

[20] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2024. Accelerating Multi-Tier Storage Cache Simulations Using Knee Detection. *Performance Evaluation* 164 (May 2024), 102410. doi:*10.1016/j.peva.2024.102410*

[21] Phyllis G. Frankl and Stewart N. Weiss. 1991. An Experimental Comparison of the Effectiveness of the All-uses and All-edges Adequacy Criteria. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV)*. ACM, Victoria, British Columbia, Canada, 154–164.

[22] Phyllis G. Frankl and Stewart N. Weiss. 1993. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Transactions on Software Engineering* 19, 8 (1993), 774–787.

[23] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I. Siminiceanu. 2009. Model-Checking the Linux Virtual File System. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, Savannah, GA, USA, 74–88.

[24] Bernhard Garn and Dimitris E. Simos. 2014. Eris: A Tool for Combinatorial Testing of the Linux System Call Interface. In *Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society Press, Cleveland, Ohio, USA, 58–67.

[25] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-adequate Test Suites using Coverage Criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Lugano, Switzerland, 302–313.

[26] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 1–33.

[27] Google. 2023. Syzkaller: Linux Syscall Fuzzer. *https://github.com/google/syzkaller*.

[28] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, Hyderabad, India, 72–82.

[29] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, Stevenson, WA, 293–306.

[30] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Vancouver, BC, Canada, 151–156.

[31] Luis Henriques and Theodore Ts'o. 2022. Ext4: Fix Error Code Return to User-space in ext4_get_branch(). *https://github.com/torvalds/linux/commit/26d75a16af285a70863ba6a81f85d81e7e65da50*.

[32] Free Software Foundation Inc. 2023. Gcov, a Test Coverage Program. *https://gcc.gnu.org/onlinedocs/gcc/Gcov.html*.

[33] Laura Inozemtseva and Reid Holmes. 2014. Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, Hyderabad, India, 435–445.

[34] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Tallinn, Estonia, 955–963.

[35] Yujuan Jiang, Bram Adams, and Daniel M. Germán. 2013. Will My Patch Make It? And How Fast? Case Study on the Linux Kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*

(MSR). IEEE, IEEE Computer Society, San Francisco, CA, 101–110.

[36] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. 2006. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*. ACM SIGOPS, Seattle, WA, 89–102.

[37] Jan Kara and Theodore Ts'o. 2022. Ext4: fix deadlock due to mbcache entry corruption. *https://github.com/torvalds/linux/commit/a44e84a9b7764c72896f7241a0ec9ac7e7ef38dd*.

[38] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Huntsville, ON, Canada, 147–161.

[39] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society Press, Montreal, QC, Canada, 560–564.

[40] Rick Kuhn, Raghu N. Kacker, Yu Lei, and Dimitris E. Simos. 2020. Input Space Coverage Matters. *Computer* 53, 1 (2020), 37–44.

[41] Hayley LeBlanc, Shankara Pailoor, Om Saran K. R. E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*. ACM, Rome, Italy, 718–733.

[42] Baokun Li, Ritesh Harjani, and Theodore Ts'o. 2022. Ext4: add inode table check in __ext4_get_inode_loc to avoid possible infinite loop. *https://github.com/torvalds/linux/commit/eee22187b53611e173161e38f61de1c7ecbeb876*.

[43] Linux Kernel Community. 2025. ftrace: Function Tracer for the Linux Kernel. *https://www.kernel.org/doc/html/latest/trace/ftrace.html*.

[44] Yifei Liu. 2024. *Towards Efficient, Scalable, and Versatile File System Model Checking*. Technical Report FSL-24-04. Computer Science Department, Stony Brook University.

[45] Yifei Liu. 2025. *Advancing File System Model Checking: Coverage, Framework, and Scalability*. Ph.D. Dissertation. Computer Science Department, Stony Brook University.

[46] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott Smolka, Wei Su, and Erez Zadok. 2024. Metis: File System Model Checking via Versatile Input and State Exploration. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST '24)*. USENIX Association, Santa Clara, CA, 123–140. *https://github.com/sbu-fsl/Metis* Received all 3 Artifact-Evaluation badges.

[47] Yifei Liu, Gautam Ahuja, Geoff Kuenning, Scott Smolka, and Erez Zadok. 2023. Input and Output Coverage Needed in File System Testing. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23)*. ACM, Boston, MA, 93–101.

[48] Yu Liu, Hong Jiang, Yangtao Wang, Ke Zhou, Yifei Liu, and Li Liu. 2020. Content Sifting Storage: Achieving Fast Read for Large-scale Image Dataset Analysis. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, 1–6. doi:*10.1109/DAC18072.2020.9218738*

[49] Yu Liu, Yangtao Wang, Ke Zhou, Yujuan Yang, and Yifei Liu. 2020. Semantic-aware data quality assessment for image big data. *Future Generation Computer Systems* 102 (2020), 53–65. doi:*10.1016/J.FUTURE.2019.07.063*

[50] Yu Liu, Yangtao Wang, Ke Zhou, Yujuan Yang, Yifei Liu, Jingkuan Song, and Zhili Xiao. 2019. A Framework for Image Dark Data Assessment. In *Proceedings of the 3rd APWeb-WAIM joint conference on Web and Big Data (APWeb-WAIM '19)*, Vol. 11641. Springer, Chengdu, China, 3–18. doi:*10.1007/978-3-030-26072-9_1* Won Best Paper Runner-Up.

[51] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '13)*. USENIX Association, San Jose, CA, 31–44.

[52] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. 2024. Monarch: A Fuzzing Framework for Distributed File Systems. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)*. USENIX, Santa Clara, CA, 529–543.

[53] Yashwant K Malaiya, Naixin Li, Jim Bieman, Rick Karcich, and Bob Skibbe. 1994. The Relationship Between Test Coverage and Reliability. In *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Monterey, CA, USA, 186–195.

[54] Filipe Manana. 2022. BTRFS: Fix NOWAIT Buffered Write Returning –ENOSPC. *https://github.com/torvalds/linux/commit/a348c8d4f6cf23ef04b0edaccdfe9d94c2d335db*.

[55] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Ottawa Linux Symposium (OLS)*, Vol. 2. Ottawa Linux Symposium, Ottawa, Canada, 21–33.

[56] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, Monterey, CA, 361–377.

[57] Subrata Modak. 2009. Linux Test Project (LTP). *http://ltp.sourceforge.net/*.

[58] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, 33–50.

[59] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2019. Crashmonkey and ACE: Systematically Testing File-System Crash Consistency. *ACM Transactions on Storage (TOS)* 15, 2 (2019), 1–34. doi:*10.1145/3320275*

[60] Akbar Siami Namin and James H. Andrews. 2009. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Chicago, IL, USA, 57–68.

[61] Thomas J. Ostrand and Marc J. Balcer. 1988. The Category-Partition Method For Specifying And Generating Functional Tests. *Commun. ACM* 31, 6 (1988), 676–686.

[62] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Broomfield, CO, 433–448.

[63] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. 2002. *System Call Clustering: A Profile-Directed Optimization Technique*. Technical Report. The University of Arizona.

[64] Ajitha Rajan. 2006. Coverage Metrics to Measure Adequacy of Black-Box Test Suites. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Tokyo, Japan, 335–338.

[65] Stuart C. Reid. 1997. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. In *Proceedings of the Fourth International Software Metrics Symposium (METRICS)*. IEEE Computer Society Press, Albuquerque, NM, USA, 64–73.

[66] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*

(SOSP). ACM, Monterey, CA, 38–53.

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

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

[69] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement.. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Savannah, GA, 1–16.

[70] Silicon Graphics, Inc. 2021. XFS – high-performance 64-bit journaling file system. *https://www.linuxlinks.com/xfs/*. Visited February, 2021.

[71] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. 2012. On the Danger of Coverage Directed Test Case Generation. In *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012*. Springer, Tallinn, Estonia, 409–424.

[72] Strace Developers. 2025. strace: Linux syscall tracer. *https://strace.io*.

[73] Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. 2021. Model-Checking Support for File System Development. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*. ACM, Virtual, 103–110. doi:*10.1145/3465332.3470878*

[74] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, Zurich, Switzerland, 386–396.

[75] Linus Torvalds. 2023. Linux Kernel Source Tree. *https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git*.

[76] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*. ACM, London, United Kingdom, 1–16.

[77] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. 2013. Semi-valid Input Coverage for Fuzz Testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Lugano, Switzerland, 56–66.

[78] Theodore Ts'o. 2022. Ext4: Fix use-after-free in ext4_xattr_set_entry. *https://lore.kernel.org/lkml/165849767593.303416.8631216390537886242.b4-ty@mit.edu/*.

[79] Matthew Wilcox and Dave Chinner. 2022. XFS: Use generic_file_open(). *https://github.com/torvalds/linux/commit/f3bf67c6c6fe863b7946ac0c2214a147dc50523d*.

[80] Darrick J. Wong and Theodore Ts'o. 2022. Ext4: don't fail GETFSUUID when the caller provides a long buffer. *https://github.com/torvalds/linux/commit/a7e9d977e031fceefe1e7cd69ebd7202d5758b56*.

[81] Darrick J. Wong and Theodore Ts'o. 2022. Ext4: dont return EINVAL from GETFSUUID when reporting UUID length. *https://github.com/torvalds/linux/commit/b76abb5157468756163fe7e3431c9fe32cba57ca*.

[82] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. KRACE: Data Race Fuzzing for Kernel File Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. IEEE, Virtual Event, 1643–1660.

[83] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. IEEE, San Francisco, CA, 818–834.

[84] Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. 2021. Test Coverage Metrics for the Network. In *Proceedings of the ACM SIGCOMM Conference*. ACM, Virtual Event, 775–787.

[85] Junfeng Yang, Can Sar, and Dawson Engler. 2006. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Seattle, WA, 131–146.

[86] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2004. Using Model Checking to Find Serious File System Errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. ACM SIGOPS, San Francisco, CA, 273–288.

[87] Duo Zhang, Om Rameshwar Gatla, Wei Xu, and Mai Zheng. 2021. A Study of Persistent Memory Bugs in the Linux Kernel. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR)*. ACM, Haifa, Israel, 1–6.

[88] Yi Zhang and Theodore Ts'o. 2022. Ext4: check and assert if marking an no_delete evicting inode dirty. *https://github.com/torvalds/linux/commit/318cdc822c63b6e2befcfdc2088378ae6fa18def*.

[89] Zhiqiang Zhang, Tianyong Wu, and Jian Zhang. 2015. Boundary Value Analysis in Automatic White-box Test Generation. In *Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society Press, Gaithersbury, MD, USA, 239–249.

[90] Ke Zhou, Yangtao Wang, Yu Liu, Yujuan Yang, Yifei Liu, Guoliang Li, Lianli Gao, and Zhili Xiao. 2020. A framework for image dark data assessment. *World Wide Web* 23, 3 (2020), 2079–2105. doi:*10.1007/S11280-020-00779-X*

[91] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys (CSUR)* 29, 4 (1997), 366–427.