Input and Output Coverage Needed in File System Testing

Yifei Liu
Stony Brook University

Gautam Ahuja
Stony Brook University

Geoff Kuenning
Harvey Mudd College

Scott A. Smolka
Stony Brook University

Erez Zadok
Stony Brook University

ABSTRACT

File systems need testing to discover bugs and to help ensure reliability. Many file system testing tools are evaluated based on their code coverage. We analyzed recently reported bugs in Ext4 and BtrFS and found a weak correlation between code coverage and test effectiveness: many bugs are missed because they depend on specific inputs, even though the code was covered by a test suite. Our position is that coverage of system call inputs and outputs is critically important for testing file systems. We thus suggest *input and output coverage* as criteria for file system testing, and show how they can improve the effectiveness of testing. We built a prototype called IOCov to evaluate the input and output coverage of file system testing tools. IOCov identified many untested cases (specific inputs and outputs or ranges thereof) for both CrashMonkey and xfstests. Additionally, we discuss a method and associated metrics to identify over- and under-testing using IOCov.

CCS CONCEPTS

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

KEYWORDS

File System Testing, Code Coverage, Input Coverage, Output Coverage

ACM Reference Format:
Yifei Liu, Gautam Ahuja, Geoff Kuenning, Scott A. Smolka, and Erez Zadok. 2023. Input and Output Coverage Needed in File System Testing. In *15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23), July 9, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. *https://doi.org/10.1145/3599691.3603405*

HotStorage '23, July 9, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0224-2/23/07…$15.00
https://doi.org/10.1145/3599691.3603405

1 INTRODUCTION

Motivation. File systems, a fundamental component of modern operating systems, must reliably store and organize user data. Due to their critical role, file system bugs are a serious matter [35, 67]. Various testing approaches have discovered such bugs and improved file system reliability [8, 38]. Testing file systems remains a challenge, however, due to their complexity, the presence of corner cases [66], their ongoing development [35], and the demand for strong resiliency (*e.g.*, crash consistency [45, 53]).

Although a number of approaches to testing file systems have been proposed and have succeeded in identifying many defects, bugs are still discovered on an almost daily basis, even in mature file systems [28, 29, 35]. This raises an important question: how can one evaluate and improve existing file system testing tools and thus find more bugs, thereby enhancing reliability?

Limitations of code coverage. Code coverage [1], the most commonly used metric for evaluating test quality [19], measures how much source code has been executed by a test suite. Coverage can be calculated at different levels, including individual lines of code, functions, and branches [23]. Although coverage is helpful in evaluating file system testing, it has two significant limitations: (1) Even though the developer knows which lines were not covered, it is challenging to modify tests to cover them [1, 5]; and (2) The code covered by tests may still hide bugs, depending on parameter values [10, 24].

To investigate the correlation between code coverage and testing effectiveness (*i.e.*, the ability to find bugs), we conducted a study (Section 2) of recent file system bugs. We found that existing testing tools are hindered by the limitations of code coverage. Moreover, the connection between test inputs (*i.e.*, system calls) and file system code is obscure [5, 15]; so improving tests to cover more code is challenging. Additionally, in our study of xfstests—one of the

oldest and most popular file system test suites [52]—53% of reported bugs involved code that xfstests covered yet failed to expose the bug. We found a similar phenomenon with other metrics such as function and branch coverage. Thus, it is imperative to find other completeness metrics that developers can use to improve test suites.

Contributions. We conducted a bug study and found that most bugs can be triggered by specific inputs (system calls and their arguments), especially near boundaries and corner cases that might be missed by some testing techniques. Therefore, we propose *input coverage* [20, 31, 60] as a file system testing metric.

Exploring input coverage alone is insufficient, however, as the same syscall input can behave differently depending on the file system state. For instance, writing to an existing file is different from writing a brand-new one. To ensure that the inputs are executed on meaningfully different states, we propose another metric, *output coverage* [3], to measure the coverage of syscall return values and error codes. This helps assess whether the testing reaches a wide variety of outputs, given that many bugs happen on exit and failure paths [35] (Section 2). Our position is that *testing techniques should include input and output coverage alongside code-coverage metrics to improve test completeness.*

To evaluate input and output coverage, we first selected 27 syscalls relevant to file systems, out of approximately 400 Linux system calls [6, 59]. Next, we inspected each selected syscall's arguments and divided them into four categories: identifiers (*e.g.*, file descriptors), bitmaps (*e.g.*, `open` flags), numeric arguments (*e.g.*, `write` size), and categorical arguments (*e.g.*, `lseek` whence). We then partitioned the input space for each type of argument and the output space for each return value (*e.g.*, the `write` buffer size was partitioned by powers of 2). We created separate partitions for boundary values and corner cases. Finally, we calculated input and output coverage by whether and how thoroughly a test suite covered those arguments and outputs.

We developed a prototype analyzer, called IOCov, to compute the input and output coverage of file system test suites. We make the following contributions:

(1) We studied patches and bugs from two popular Linux file systems and investigated the correlation between code coverage and bug-finding effectiveness of xfstests. We also identified common triggers of file system bugs; this was not addressed in previous studies [35, 67].
(2) We studied syscalls to define their input and output coverage, and used that analysis to evaluate file system testing methods and to discover and overcome their code-coverage limitations.
(3) We designed IOCov to accurately measure the input and output coverage of existing file system test suites.
(4) We empirically evaluated the input and output coverage for two representative file system test suites, xfstests [52] and CrashMonkey [41], and found many untested regions for both.

2 REAL-WORLD BUG STUDY

Code coverage effectiveness. Although many researchers have studied the correlation between code coverage and test effectiveness, they usually focused on small programs [10, 21, 43] or relatively simple user applications [16, 24]. To the best of our knowledge, there is no existing work that considers this correlation for in-kernel file systems based on real-world bugs and test suites. Here, we discuss the findings from a study we conducted on two popular Linux file systems: Ext4 [37] and BtrFS [50]. First, because of the strong link between Git commits and *accepted* patches [25, 57], we manually analyzed the latest 100 Git commits from the Linux kernel repository [58] applied in 2022 for each file system—200 commits in total. Second, using Lu *et al.*'s technique [35], we identified which of the 200 commits were bug fixes. This identified 51 Ext4 bugs and 19 BtrFS bugs. We found fewer bugs for BtrFS because many commits were due to a major code refactoring in December 2022. Third, we ran xfstests on Ext4 and BtrFS with all the generic and file-system–specific tests and recorded code coverage, including line, function, and branch coverage. For each bug fix, we examined whether xfstests covered the pertinent code, and whether the suite detected the bug. This approach allowed us to study the correlation between bug-detection ability and code coverage in xfstests. Finally, we analyzed the syscalls required to trigger these bugs and code paths of each bug.

We used `Gcov` [23] to compute the code coverage of xfstests on Linux kernel v6.0.6. For each bug-fix commit, we manually inspected `Gcov`'s report to determine whether the buggy code was covered. Since our bug study was manual, two people independently cross-validated all findings. We found that for 37 out of 70 bugs (53%), xfstests covered the relevant code lines but still missed the bugs. Moreover, xfstests missed bugs in 61% (43 out of 70) of covered functions and 29% (20 out of 70) for covered branches. We conclude that code-coverage metrics are not strongly correlated with test effectiveness (*i.e.*, the ability to find bugs).

Bug Classification. Next, we manually inspected each bug-fix commit from the perspective of software testing and analyzed the factors that triggered the bug in question. We observed that most bugs could be detected only with specific syscall inputs, which we characterized as *input bugs*. Another finding is that many bugs occur on the exit path; such bugs may alter the behavior of syscall returns. We defined these as *output bugs*. We analyzed each bug to determine its classification as an input bug, output bug, both, or neither.

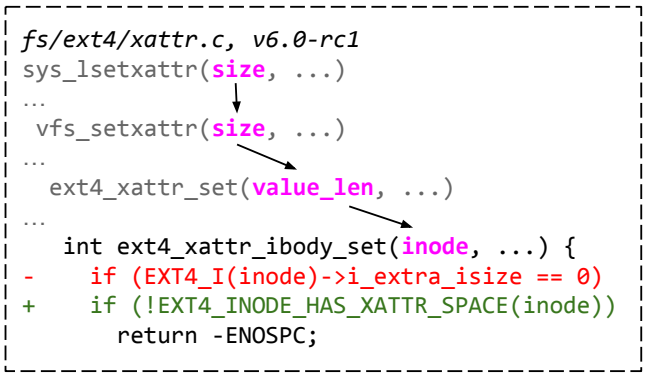

Figure 1: An example of a both input-related and output-related Ext4 bug. The bug was fixed by checking whether the inode has room to store additional xattrs in `ext4_xattr_ibody_set`**.**

We found that a major proportion (71%, 50 out of 70) of the bugs were input bugs; also, 59% of bugs (41 out of 70) appeared in exit paths that affect syscall returns [22, 36]. Altogether, 57 out of 70 bugs (81%) were related to syscall inputs or outputs. Among the bugs in covered code that were missed by xfstests, 24 out of 37 (65%) could be triggered by specific syscall arguments, indicating that input coverage can compensate for the shortcomings of code coverage. Specifically, these arguments frequently involved corner cases [7], less-tested inputs [32], and boundary values [61]; these are usually ignored by code-coverage metrics because they often execute the same code as heavily-tested inputs [59].

Figure 1 shows such an example from a recent bug [61] in Ext4 that involves both input and output. This bug's lines, function, and branches are all covered by xfstests, which nevertheless failed to find it because it happened only when `lsetxattr` used the maximum allowed `size` argument, causing the minimum offset (`min_offs`) between two block groups to overflow. As this bug is also an output bug, a file system tester could detect it by checking the correctness of the condition for the error case (*i.e.*, `ENOSPC`). In sum, covering code alone is not enough for finding bugs because many bugs depend on specific inputs and outputs.

We will make the bug study dataset publicly available, including the code-coverage analysis and triggers for each bug, as well as the classification of input and output bugs.

3 IOCOV FRAMEWORK

We define input and output coverage by partitioning those spaces, and describe how the IOCov framework computes input and output coverage for file system test suites.

Input- and output-space partitioning. Our bug study confirmed the importance of thoroughly covering test inputs and outputs, so we wanted to define metrics to measure that coverage. Linux has around 400 syscalls [6, 59]. It is impractical to measure test adequacy for all of them, so we focused on the core file-system-related syscalls. Still, the input space is large because most syscalls take multiple arguments with arbitrarily large values. Thus, we partitioned each argument's input space to identify the partitions that are under- or over-tested [48]. We divided arguments into four classes: identifier, bitmap, numeric, and categorical. Identifiers include file descriptors and path names. Bitmaps can be logically ORed (*e.g.*, `open` flags or `chmod` permissions). Numeric arguments often represent a number of bytes (*e.g.*, `write` size). Categorical arguments have fixed available values (*e.g.*, `lseek`'s *whence*).

We used different methods to partition each argument type. For bitmaps we considered each flag and certain combinations thereof. For numeric arguments, we considered boundary-value analysis [12, 44, 48, 68], but ultimately used powers of 2 as boundaries because they are common in file systems [26]. Most syscall outputs return either success or an error code, so we partitioned the output space on success vs. failure, and further by each error code. For syscalls that return a byte count on success (*e.g.*, `write`), we partitioned successful returns by powers of 2.

Input and output coverage. Next, we defined *input coverage* and *output coverage* as how much a tester exercises an argument's input or output partitions; the latter also indirectly measures how well error codes are exercised, since many bugs happen on error paths. We note that some errors are harder to trigger than others. For example, triggering `ENOMEM` requires a system with limited memory. Therefore, achieving 100% coverage of all errors may be challenging. Nevertheless, using input- and output-coverage metrics, developers can compare and improve file system test suites more easily than by considering code coverage alone, because: (1) our metrics more directly identify any missed or under-tested inputs or outputs, and (2) code-coverage metrics require going through complex kernel call stacks [5].

IOCov implementation. Our prototype IOCov measures the input and output coverage of existing file system testers by tracing syscalls with LTTng, a low-overhead tracing framework [2, 34]. Traced syscalls and their arguments are sent to the IOCov analyzer, which analyzes them and calculates coverage metrics. IOCov has three components: the trace filter, the syscall variant handler, and the input/output partitioner.

Most file system testers use dedicated devices and mount points for testing (*e.g.*, `/mnt/test` for xfstests). Since LTTng records all syscalls from the file system tester, it observes other syscalls that are not directly used to test a file system. We therefore developed a set of regular expressions to filter out those irrelevant system call records (*e.g.*, based on the mount point pathname) before IOCov analyzes them further.

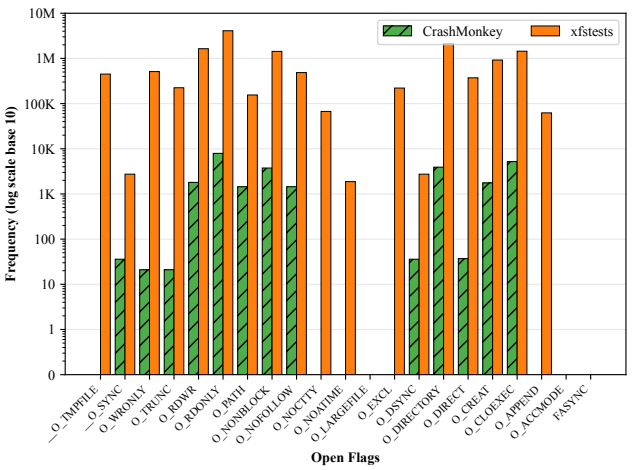

Figure 2: Input coverage of `open` flags for CrashMonkey and xfstests. The x-axis lists all possible flags supported by `open`. The y-axis (log_{10}) shows the frequency of each `open` flag exercised by each testing tool.

Test Suite / % for #flags	1	2	3	4	5	6
CrashMonkey: all flags	9.3	2.8	22.1	65.4	0.5	0
CrashMonkey: O_RDONLY	9.3	2.8	21.9	65.6	0.5	0
xfstests: all flags	6.1	28.2	18.2	46.8	0.5	0.4
xfstests: O_RDONLY	6.0	30.8	10.5	51.9	0.5	0.3

Table 1: Percentage of time that 1–6 `open` flags were used together, for CrashMonkey and xfstests. The table header numbers indicate how many flags were combined in `open` for testing (where "1" means a single flag used alone). Because O_RDONLY is the most popular flag, we also analyze all flag combinations that included that flag.

Many syscalls have variants with different prototypes (*e.g.*, `open`, `openat`, `creat`, and `openat2`). Variants share almost the same kernel implementation [47, 59], so IOCov's variant handler merges their input and output spaces when computing coverage. Lastly, the input/output partitioner divides the input and output spaces, counts the occurrences of each partition, and calculates coverage metrics. IOCov is easy to use. The only setting that needs to be adjusted when applying it to a new file system tester is the regular expression used to identify the tester's mount points.

4 EVALUATION

We experimented with the IOCov prototype on two file system testers: CrashMonkey [41] and xfstests [52]. The test machine had 4 cores and 128GB RAM. CrashMonkey is an automatic black-box tester for file system crash consistency; xfstests is a hand-written regression test suite. We tested Ext4 with all CrashMonkey's tests (including all of seq-1's 300 workloads and all generic tests) as well as all of the 706 generic tests and 308 Ext4-specific tests from xfstests.

Currently, IOCov measures input coverage for 14 distinct arguments from a total of 27 syscalls, including 11 base syscalls (`open`, `read`, `write`, `lseek`, `truncate`, `mkdir`, `chmod`, `close`, `chdir`, `setxattr`, and `getxattr`) and their variants; it also records output coverage for all 27 syscalls.

Input coverage results. Figure 2 shows the input coverage of `open`, partitioned by individual flags, for CrashMonkey and xfstests. The x-axis labels all possible flags supported by `open`. The y-axis (log_{10}) shows how often each `open` flag was exercised by the testing tool. A higher y-value corresponds to more frequent usage of a particular `open` flag by a test suite. For instance, `O_RDONLY`, which is universally applied to open a file as read only, is the most-used flag for both CrashMonkey and xfstests. Figure 2 shows that CrashMonkey and xfstests used the `O_RDONLY` flag $7,924$ and $4,099,770$ times, respectively.

The `open` flag frequency of xfstests is larger than CrashMonkey's for every flag, showing that xfstests tests them more thoroughly. We can see that some flags are not tested at all; this information can help developers identify new tests (*e.g.*, bugs exist for O_LARGEFILE [62]). We also analyzed the number of tested combinations of flags. Table 1 shows that both suites used at most six `open` flags together. In this table, "All" denotes all instances of `open` flags, and "O_RDONLY" limits the results to instances with that (most popular) flag. Using four flags was the most common. For CrashMonkey, the second most frequent combination was three flags; for xfstests it was two. This highlights the different strategies used by the two test suites and suggests that more diversified test cases can be designed to test more `open` flag combinations.

Figure 3 shows the input coverage of the `write` size parameter (*i.e.*, requested byte count). The x-axis shows the log_2 of the size. Because we use powers of 2 as boundary values, each interval (*i.e.*, input-space partition [4]) along the x-axis includes the actual `write` sizes rounded down to the nearest lower boundary value. For example, $x = 10$ represents all write sizes from 2^{10} to $2^{11} - 1$ (or 1024–2047). The x-axis also includes a special "Equal to 0" value (unusual but allowed under POSIX [56]). The size 0 is also a boundary value because it is the minimum possible size accepted by `write` but is easily neglected by testing [48]. The y-axis (log_{10}) of Figure 3 shows the frequency of each x value.

The `write` size frequency of xfstests is larger than CrashMonkey's for every interval. CrashMonkey did not exercise many write sizes, and neither tool tested any sizes over 258 MiB (annotated in Figure 3) despite the fact that 64-bit

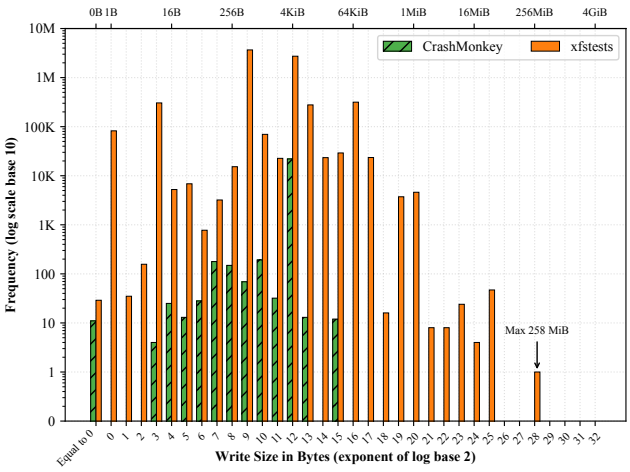

Figure 3: Input coverage of `write` size (in bytes) for CrashMonkey and xfstests. The x-axis shows the log_2 of write size, rounded down to the nearest boundary value. The $x2$-axis shows the actual size corresponding to the x-axis. For example, $x = 28$ represents 2^{28} or 256MiB. The y-axis (log_{10}) shows the frequency.

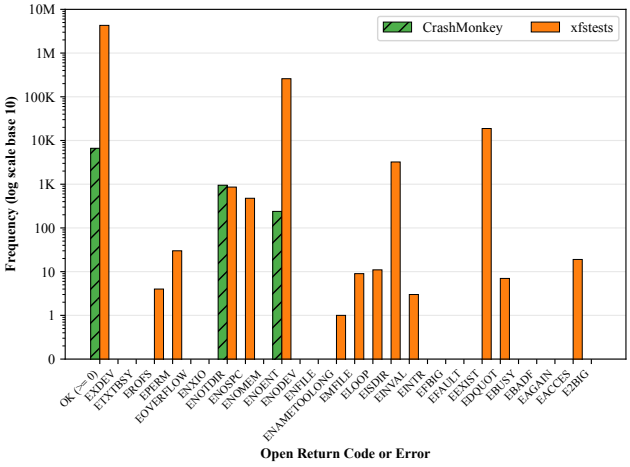

Figure 4: Output coverage of `open` for CrashMonkey and xfstests. The x-axis shows outputs (*i.e.*, success and error codes) returned by `open` and its variants. The y-axis (log_{10}) shows the frequency.

systems with many GB of RAM are common (and the maximum Ext4 file size is 16TB). Due to space limitations, we do not show input coverage for other IOCov-supported syscalls. Overall, we found untested input partitions for many syscalls; xfstests generally has better coverage than CrashMonkey.

Output coverage results. Figure 4 shows the output coverage for `open` syscalls. The x-axis shows all possible error codes returned by `open` and its variants. The y-axis (log_{10}) shows the frequency of each output partitioned by success and error codes. Notably, we obtained the error codes appearing along the x-axis from the `open` manual page, which may not be consistent with the actual implementation. "OK" means any return value that is ≥ 0 (*i.e.*, `open` succeeded). The xfstests suite covered more error cases than CrashMonkey except for ENOTDIR. Still, many possible error codes remain untested.

Application: syscall test adequacy. The above figures show how much coverage each test suite had for each partition; but we also wanted to offer a *single* metric that can numerically represent the test adequacy for each input and output. We observed that some partitions are tested millions of times while others are not tested at all. Thus, we introduced the notions of under-testing and over-testing for each partition. Under-testing means that the partition gets too little testing if at all; this can miss bugs. Over-testing means the partitions are excessively tested; this could waste resources better diverted elsewhere (*e.g.*, under-tested partitions). We note that assessing the appropriate amount of testing may depend on the partition itself: for example, smaller `write` sizes are more common and may benefit from more testing than large ones. Moreover, we wanted our metric to "penalize" under-testing as well as possible over-testing.

Thus, we define a *Test Coverage Deviation (TCD)* metric as our first attempt to evaluate how comprehensive are the input and output coverage values. Given an input or output coverage for a syscall with N partitions, where the frequency for partition i is F_i, we first define a target array T of length N, where T_i is the number of times (frequency) we want to test partition i. The TCD for the array T is the Root Mean Square Deviation (RMSD [27]):

$$TCD_T = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log F_i - \log T_i)^2}$$

We use logarithms for the frequencies and target because under-testing is more problematic than over-testing [33], so we want to downplay the latter. A lower TCD is better because it is closer to the pre-defined test target T. The selection of T depends on file system developers' preferences. For example, crash-consistency testing heavily exploits persistence operations [41], such as `sync` or the `O_SYNC` flag of `open`. Thus, developers might want to set a larger target T_i for persistency-related input or output partitions. IOCov can be used to evaluate TCD iteratively; this can help developers design test cases that avoid under- or over-testing of the desired inputs and outputs.

For simplicity, in our study we set all elements of the array T to the same value. Figure 5 shows the TCD for CrashMonkey and xfstests with different target arrays. The x-axis

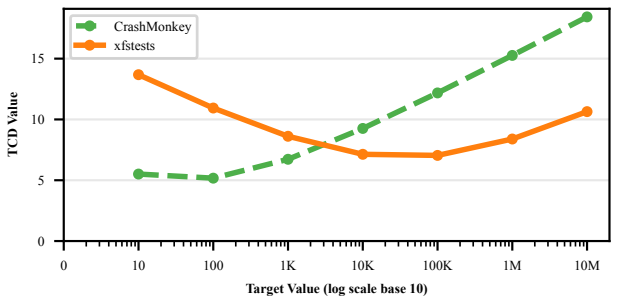

Figure 5: Test Coverage Deviation (TCD) for open flags, for CrashMonkey and xfstests. The x-axis (log_{10}) shows the target number of tests for each open flag. The y-axis shows the TCD value of each testing tool for different target values.

(log_{10}) shows the uniform value of the target array for open flags. We see that below x ≈5,237, CrashMonkey has a better (lower) TCD; above that value, xfstests is better. This matches Figure 2, where CrashMonkey generally had lower test frequencies for open flags.

Our TCD metric accounts for both under-testing and over-testing, and provides developers with a more comprehensive view of the test suite's adequacy for a given target, allowing developers to optimize test strategies and effectiveness.

5 RELATED WORK

Test coverage metrics. The correlation between code coverage and test effectiveness has been well studied, but the strength of the correlation varies depending on test targets [16, 17, 30] or subclass metrics [13, 14, 19, 21]. Gopinath *et al.* [19] stated that statement coverage is best at finding faults, but Hemmati *et al.* [21] found it weaker than other metrics (*e.g.*, branch coverage) for the same task. However, other work [9, 10, 24, 43] showed that code coverage has a low-to-moderate correlation with test effectiveness, and thus new, complementary coverage criteria are needed [54]. Still, no existing research considers the correlation for complex low-level software like in-kernel file systems. Some research proposed input-coverage concepts [20, 31, 60] but did not offer syscall metrics and is not applicable to file-system testing. The input and output coverage we propose for file system testing can also apply to other testing tasks like database testing [39], where the syscall inputs and outputs are transformed into inputs and outputs of database queries.

File-system testing. Regression-testing suites such as xfstests [52] and LTP [40] use hand-written tests for various aspects of file system functionality. It is difficult for hand-written tests to guarantee thorough coverage of inputs and outputs. Model checking [15, 42, 49, 55, 65, 66] compares the file system implementation with a specification and searches for mismatches. Although it can check many corner states, model checking is slow (especially for I/O-bound storage systems) and has a state-explosion problem.

Black-box testing [11, 41] generates rule-based syscall workloads, but does not ensure full exploration of input and output spaces. Finally, fuzzing [18, 29, 51, 63, 64] stresses file systems by input mutation to maximize path coverage, but path coverage (*i.e.*, subtype of code coverage) has drawbacks—missing bugs—similar to code-coverage methods [19, 24].

6 CONCLUSION AND FUTURE WORK

In this paper, we studied real file system bugs and identified the limitations of code coverage and the importance of covering diverse syscall inputs and outputs. This motivated us to propose input and output coverage for file system testing and implement IOCov to measure this coverage for existing file system testing tools. Our preliminary results show that CrashMonkey and xfstests fail to test many input and output cases; this information can be readily used to improve these testing tools. We also proposed and analyzed a new metric, Test Coverage Deviation (TCD), to evaluate and compare the amount of under- and over-testing of file system test tools.

Future work. We plan to support more syscalls, enhance our metrics to support bit combinations, explore non-uniform target arrays (T), and support file descriptors and pointer arguments. We also plan to evaluate fuzzing systems [18, 29, 63, 64]. For different fuzzers, IOCov needs to apply other techniques to trace fuzzed syscalls. For example, Syzkaller [18] logs syscalls with declarative descriptions, which need to be parsed by IOCov. Hydra [29, 64], however, exercises syscalls with Library OS [46], so IOCov requires a different method than LTTng to trace syscalls.

We are currently developing a differential-testing-based file system tester utilizing IOCov. Our approach has found several new bugs that we fixed and reported; one has already been merged into the Linux mainline.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We are also grateful to Dongyoon Lee for his valuable assistance in helping us design the new test coverage metric. 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-2106434, CNS-2214980, CPS-1446832, ITE-2040599, and ITE-2134840.

REFERENCES

[1] Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. 2011. Code Coverage Analysis in Practice for Large Systems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, Waikiki, Honolulu, HI, USA, 736–745.

[2] 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.

[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] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press, Cambridge, United Kingdom.

[5] 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.

[6] 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.

[7] 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*.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] Nikolas Havrikov, Alexander Kampmann, and Andreas Zeller. 2022. From Input Coverage to Code Coverage: Systematically Covering Input Structure with k-Paths. (2022), 42 pages. In submission.

[21] 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.

[22] 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*.

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

[24] 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.

[25] 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.

[26] 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.

[27] Wolfgang Kabsch. 1976. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography* 32, 5 (1976), 922–923.

[28] Kernel.org Bugzilla. 2023. Ext4 Bug Entries. *https://bugzilla.kernel.org/buglist.cgi?component=ext4*.

[29] 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, Toronto, Ontario, Canada, 147–161.

[30] 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.

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

[32] Jerry Lee and Theodore Ts'o. 2022. Ext4: Continue to Expand File System When the Target Size Doesn't Reach. *https://github.com/torvalds/linux/commit/df3cb754d13d2cd5490db9b8d536311f8413a92e*.

[33] David Leon and Andy Podgurski. 2003. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing

Test Cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Denver, CO, USA, 442–453.

[34] LTTng. 2019. LTTng: an open source tracing framework for Linux. (April 2019). *https://lttng.org*.

[35] 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.

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

[37] 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.

[38] 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.

[39] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating Targeted Queries for Database Testing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, Vancouver, BC, Canada, 499–510.

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

[41] 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.

[42] Madanlal Musuvathi, Andy Chou, David L. Dill, and Dawson R. Engler. 2002. Model Checking System Software with CMC. In *Proceedings of the 10th ACM SIGOPS European Workshop*. ACM, Saint-Emilion, France, 219–222.

[43] 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.

[44] 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.

[45] 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.

[46] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux Kernel Library. In *9th RoEduNet IEEE International Conference*. IEEE, Sibiu, Romania, 328–333.

[47] 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.

[48] 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.

[49] 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.

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

[51] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. USENIX Association, Vancouver, BC, Canada, 167–182.

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

[53] 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.

[54] 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.

[55] 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 (HotStorage '21)*. ACM, Virtual, 103–110. *https://doi.org/10.1145/3465332.3470878*

[56] The Linux man-pages project 2023. *write(2) - Linux manual page*. The Linux man-pages project. *https://man7.org/linux/man-pages/man2/write.2.html*.

[57] 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.

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

[59] 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.

[60] 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.

[61] 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/*.

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.