Towards Efficient, Scalable, and Versatile File System Model Checking

A Dissertation Proposal presented

by

Yifei Liu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-24-04

November 2024

Stony Brook University
The Graduate School

Yifei Liu

We, the thesis committee for the above candidate for the
degree of Doctor of Philosophy, hereby recommend
acceptance of this thesis proposal

Erez Zadok - Dissertation Advisor
Professor, Computer Science Department

Omar Chowdhury - Chairperson of Dissertation Proposal
Associate Professor, Computer Science Department

Scott A. Smolka
Professor, Computer Science Department

Geoff Kuenning
Emeritus Professor, Department of Computer Science, Harvey Mudd College

Abstract of the Dissertation Proposal

Towards Efficient, Scalable, and Versatile File System Model Checking

by

Yifei Liu

Doctor of Philosophy

in

Computer Science

Stony Brook University

November 2024

Developing and maintaining robust file systems is challenging. Despite the invention of many file-system testing techniques, new bugs continue to emerge. Worse, new file systems are often less tested and present challenges for existing testing methodologies. This requires not only an effective test-metric technique to enhance existing tools but also a new approach for thoroughly checking file systems.

This thesis proposal has three thrusts. In the first thrust, we analyze recently reported bugs in Linux file systems. We discovered a weak correlation between code coverage and test effectiveness. We also observed that many file-system bugs occur along specific inputs and error paths. We thus propose input and output coverage as criteria for file system testing. We developed a prototype coverage analyzer called IOCov to compute the input and output coverage of file system testing tools. By evaluating input and output coverage for several existing testing tools, IOCov identified many untested cases. Additionally, we present a method and associated metrics to identify over- and under-testing using IOCov.

In the second thrust, we present Metis, a model-checking framework designed for versatile and thorough file system testing in the form of input and state exploration. Metis uses a nondeterministic loop and a weighting scheme to decide which system calls and their arguments to execute. Metis features a new abstract

state representation for file-system states in support of efficient state exploration. While exploring states, Metis compares the behavior of the file system under test against a reference file system and reports any discrepancies. To speed up the model-checking process, we also developed RefFS, a small, fast file system that serves as a reference, with special features designed to accelerate model checking. Experimental results show that the rate at which Metis explores file-system states scales nearly linearly across multiple compute nodes. RefFS explores states 3–28$\times$ faster than other, more mature file systems. Metis aided the development of RefFS, reporting 11 bugs that were subsequently fixed. Metis further identified 15 bugs from seven other file systems, six of which were confirmed, and one was fixed and integrated into mainline Linux.

We propose enhancing IOCov by supporting more syscalls and arguments and applying it to existing file system testing tools. We plan to extend Metis to more extensively evaluate its performance and scalability and test a broader range of file systems. We also aim to enhance the scalability of Swarm verification when applied to Metis by utilizing containers and orchestrators as our third thrust.

It is our thesis that file system model checking can be improved to achieve comprehensive input and state coverage, and to efficiently check file systems with fewer constraints, ultimately detecting bugs more quickly and improving file system reliability. Also, file system testing should include input and output coverage to improve test completeness and effectiveness. Moreover, file system model checking should scale across multiple nodes to enhance performance and scalability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

File systems are a crucial component of operating systems, serving as the backbone of the modern storage hierarchy and supporting a wide range of applications including databases [17], cloud storage [84], big data processing [85], and virtualization technologies [90]. Their dependability and robustness directly impact the overall system's reliability, making thorough testing of file systems essential to ensure data integrity, fault tolerance, and system stability [114, 123, 82]. Although many file system testing techniques have been developed, new bugs continue to surface [70]. Compounding the issue is the emergence of new file systems that often receive limited testing, posing challenges to traditional techniques [137, 30].

There are two orthogonal approaches to better address these challenges, both of which work to improve file system reliability and reduce bugs. First, given the abundance of existing testing techniques, coverage metrics (*e.g.*, code coverage and other coverage metrics) can be used to assess and enhance these methods in order to uncover more hidden bugs [72]. This requires the development of new, effective coverage metrics [59] designed specifically for file system testing. Second, a new technique for testing file systems needs to be developed to achieve more comprehensive test cases and coverage for uncovering bugs, while imposing fewer restrictions (*e.g.*, requirement of kernel instrumentation or modification) on testing newly emerging file systems [137, 70].

Nevertheless, both approaches require further improvement. Regarding coverage metrics, most testing tools employ code coverage to assess test completeness. Despite its prevalence, the effectiveness of code coverage in file system testing remains under-investigated [83]. Additionally, even though the developer knows which lines were not covered, it is challenging to modify tests to cover them [7, 2].

Different tools have been developed for testing file systems using various ap-

proaches, but new bugs (both in-kernel and non-kernel) continue to emerge on a regular basis [70, 140, 69]. The limitations of these tools stem from three main factors: coverage [7], applicability [140], and the effectiveness of bug-finding checkers [141]. In terms of coverage, an effective testing tool should handle corner cases, including a wide range of file system operations and uncommon file system states [142]. Given the variety of file systems being created, it is essential that testing tools are applicable across these different file systems. There are various types of bugs in file systems, and finding them requires using different checkers (such as POSIX violation checkers [114] and crash consistency checkers [98]). Therefore, if the checkers used in testing are inappropriate or ineffective, it may result in the inability to detect hidden bugs.

In this thesis, we propose addressing these challenges in file system testing coverage metrics and conducting more thorough and unrestricted checks on file systems through three main thrusts: (1) designing new coverage metrics that enhance file system testing, based on insights from a real-world bug study; (2) developing a file system model checking framework for comprehensive input and state exploration to effectively check emerging file systems; and (3) improving the scalability of Swarm model checking using containers and orchestrators.

We address the first challenge by starting with a study of existing file system bugs to evaluate the effectiveness of the most widely used coverage metric for file system testing: code coverage [2]. Our findings indicate a weak correlation between code coverage and test effectiveness. Based on an analysis of bugs in Ext4 and BtrFS, we propose two new coverage metrics, input coverage and output coverage, to evaluate file system testing tools by examining the coverage of syscall inputs and outputs. We also developed a prototype tool, IOCov, to compute the input and output coverage of file system testing tools effectively. We demonstrate how IOCov can be utilized to identify many untested cases, such as specific inputs, outputs, and their ranges, in existing file system testing tools, along with revealing over- and under-testing problems, providing valuable insights for further improvement.

To address the second challenge, we created Metis, a novel model-checking framework that enables thorough and versatile input and state space exploration of file systems. Metis utilizes the extensive state space exploration capabilities of model checking, combined with cross-implementation validation underlying differential testing, to check file systems without the need to construct an abstract model. This approach enables users to check file systems without requiring extensive knowledge in file systems and model checking, and simplifies the process of checking new file systems, as there is no need to create a model for each newly

developed file system. We also developed RefFS, a lightweight and fast reference file system to accelerate model checking and improve bug reproducibility with innovative ioctl APIs. To handle the vast file system input and state space, Metis supports parallel and distributed exploration using Swarm verification techniques [55] across multiple cores and machines. Metis has successfully identified over 15 bugs across various file systems, and its replayer played a crucial role in reproducing and confirming these detected bugs.

We plan to further explore the advantages of input and output coverage in file system testing. Specifically, we aim to use IOCov to enhance input coverage in existing testing tools like CrashMonkey [98], and leverage the improved tools to identify new file system bugs. We believe that the input and output coverage metrics can be easily and practically used to refine testing tools, and the benefits of input coverage and Metis can be adopted by other tools as well. We also plan to enhance Swarm verification using containers and orchestration techniques [93], referred to as Containerized Swarm Verification (CoSV), as the third thrust of this thesis proposal. CoSV will enhance the scalability of Swarm verification, enabling it to scale across more nodes, hybrid clouds, and heterogeneous hardware and software environments. This approach benefits not only Metis but also other SPIN-based model checking tasks. CoSV also provides other benefits, including environment isolation and consistency, fine-grained resource allocation, simplified deployment and orchestration, as well as improved portability.

It is our thesis that input and output coverage metrics are required for file system testing, and new model checking techniques should be used to check file systems more thoroughly with fewer restrictions. We first introduce new coverage metrics to evaluate and improve file system testing, along with a system to efficiently compute these metrics. We then develop a model-checking framework and a reference file system to enable more thorough and effective testing of file systems. Moreover, using containers for Swarm verification is a promising method to improve scalability and explore larger state spaces.

The rest of this thesis proposal is organized as follows. Chapter 2 outlines our thesis statement and describe our motivation. Chapter 3 discusses related works. Chapter 4 describes our input and output coverage metrics and the IOCov framework. Chapter 5 describes the Metis model checking framework and the RefFS reference file system. Chapter 6 describes our proposed future work including CoSV and IOCov enhancement, and Chapter 7 concludes this thesis proposal.

Chapter 2

Motivation

In this chapter, we describe our vision and our motivations behind developing new coverage metrics for file system testing and a new file system model checking framework.

2.1 Thesis Statement

The problems we aim to address in this thesis proposal are twofold: (1) discovering and computing effective coverage metrics for file system testing that make bug detection easier, and (2) developing a new file system model-checking approach that overcomes the limitations of existing tools, such as inadequate coverage and difficulty in adapting to emerging file systems. This thesis proposal introduces input and output coverage metrics for file system testing, along with the IOCov framework to address the first problem, and the Metis model-checking framework to tackle the second problem. The goal of this thesis proposal is to streamline the entire process of file system testing and model checking, assist developers in identifying hard-to-detect file system bugs and defects, and improve file system reliability across multiple dimensions.

2.2 Coverage Metrics for File System Testing

Software testing requires coverage metrics to measure effectiveness, ensure comprehensive testing, identify untested components, and improve and prioritize testing efforts [67]. File system testing is no exception. Various types of coverage metrics are applied in file system testing. For example, regression test suites

(*e.g.*, xfstests [118]) typically focus on functionality coverage, seeking to test as many functions and features of the file system as possible. Some black-box testing tools [98, 20], while not guided by an explicit coverage metric, still manage to achieve comprehensive coverage of syscall combinations. Among various metrics, code coverage and its variants are the most widely used in file system testing [43, 138, 70]. However, despite the prevalence of code coverage, its effectiveness in file system testing has not been well studied. Effectiveness of coverage metrics can be described in multiple ways. In this context, we define it as the ability to assist in identifying bugs. For example, the effectiveness of code coverage should be measured by its capacity to detect hidden bugs within the covered code.

In addition to effectiveness, another important dimension for evaluating a coverage metric is how easily developers can use it to improve their testing tools, which we refer to as the usability of coverage metrics [2]. However, due to the complexity of file systems, the utility of coverage metrics presents an additional challenge in evaluating and improving file system testing tools. In this thesis proposal, we explore the effectiveness and usability of code coverage and introduce new coverage metrics that offer both high effectiveness and usability in file system testing.

2.3 File System Model Checking

Model checking is an automated technique used to verify finite-state concurrent systems by exhaustively exploring a bounded state space to determine if the system's model adheres to its specification [24]. Model checking is well-suited for file systems due to their complex operations (*e.g.*, links, renaming), their use of diverse storage devices, and their execution under extreme conditions (*e.g.*, disk failures, crashes). These factors generate a wide range of file system states, including many corner cases [142, 141]. Model checking excels at exploring all possible states and transitions, ensuring that even rare or hard-to-reach states are checked, minimizing the chance of missing subtle bugs [25]. Given this, model checking has been successfully applied to identify numerous file system bugs and improve overall reliability. However, several challenges still need to be addressed in file system model checking, including: (1) the difficulty in building an abstract file system model, (2) limitations in detecting non-crash bugs, and (3) challenges in exploring large state spaces.

In this thesis proposal, we aim to address the limitations of existing file system model checking and fully explore the potential of model checking techniques.

Specifically, unlike conventional model checking, we do not manually create models for file systems, as they are too complex to construct accurate, practical, and universally applicable models. We adopt the approach of implementation-level model checking [47] to overcome the limitations of existing methods: eliminating the need for users to manually create checkers, enabling the detection of a wide range of bug types, and simplifying the process of checking emerging file systems.

Chapter 3

Related Work

In this chapter, we survey related works about test coverage metrics, file system testing and debugging, and verified file systems.

3.1 File System Test Metrics

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 [72, 39, 40] or subclass metrics [33, 34, 44, 51]. Gopinath *et al.* [44] stated that statement coverage is best at finding faults, but Hemmati *et al.* [51] found it weaker than other metrics (*e.g.*, branch coverage) for the same task. However, other work [102, 15, 16, 59] showed that code coverage has a low-to-moderate correlation with test effectiveness, and thus new, complementary coverage criteria are needed [122]. Still, no existing research considers the correlation for complex low-level software like in-kernel file systems. Some research proposed input-coverage concepts [50, 128, 74] 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 [96], where the syscall inputs and outputs are transformed into inputs and outputs of database queries.

File-system testing Regression-testing suites such as xfstests [118] and LTP [97] 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 [100, 142, 141, 123, 37, 114] compares the file system implemen-

tation 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 [98, 20] generates rule-based syscall workloads, but does not ensure full exploration of input and output spaces. Finally, fuzzing [43, 140, 70, 138, 117] 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 [59, 44].

3.2 File System Testing and Verification

File system testing and debugging We divide existing file system testing and bug-finding approaches into five classes: Traditional Model Checking, Implementation-level Model Checking, Fuzzing, Regression Testing, and Automatic Test Generation. Table 5.3 summarizes these approaches across various dimensions.

Traditional model checking [37, 143] builds an abstract model based on the file system implementation and verifies it for property violations. Doing so demands significant effort to create and adapt the model for each file system, given the internal design variations among file systems [87].

Implementation-level model checking [142, 141] directly examines the file system implementation, eliminating the need for model creation. Due to file systems' complexity, however, this approach requires either intrusive changes to the OS kernel [142, 141] or manually crafting system-specific checkers [141]. Additionally, existing work [142, 141] based on this approach generally only identifies crash-consistency bugs and is incapable of detecting silent semantic bugs.

Unlike these methods, Metis checks file systems for behavioral discrepancies on an unmodified kernel. Thus, there is no need to manually create checkers when testing a new file system [141]. Moreover, other model-checking approaches rely on fixed test inputs [37, 141] and lack the versatility to accommodate different input patterns. All model-checking approaches, including Metis, track file system states to guarantee thorough state exploration [24], a feature often lacking in other approaches.

Model checking and fuzzing are orthogonal approaches, each with its own advantages and disadvantages. File system fuzzing [43, 138, 70, 140] continually mutates syscall inputs from a corpus, prioritizing those that trigger new code coverage for further mutation and execution, but they cannot make state-coverage guarantees, risk repeatedly exploring the same system states, and require ker-

nel instrumentation. Some fuzzing techniques [70, 140] also corrupt metadata to trigger crashes more easily and use library OS [109] to achieve faster and more reproducible execution than VM-based fuzzers. However, such designs have their own drawbacks: they require file-system–specific utilities to locate metadata blocks and cannot test out-of-tree file systems unsupported by library OS. Hybridra [144] enhances existing file system fuzzing with concolic execution, but it remains fuzzing-based and has the same limitations of file system fuzzers, including the lack of state-coverage guarantees.

Fuzzing mainly supplies inputs to stress file systems and commonly finds bugs using external checkers, such as KASan [42] (memory errors) and SibylFS [114] (POSIX violations). Current fuzzers configure the tested syscalls but not their arguments [43, 117], as testing is driven by code coverage. Compared to fuzzing, Metis employs a test strategy that explores both the input and state spaces, rather than solely maximizing code coverage.

Manually written regression-testing suites like xfstests [118] and LTP [97] check expected outputs and ensure that code updates do not [re]introduce bugs. Because they are hand-created, they are not easily extensible and do not attempt to automate or systematize their input or state exploration. Compared to their XFS-specific tests, xfstests' "generic" tests can be used with any file system. Nevertheless, from our past experience (including building RefFS), even when adopting the generic tests, some setup functions must be manually modified.

Automatic test generation [98, 20, 76] creates rule-based syscall workloads (*e.g.*, opening a file before writing) and employs external checkers (*e.g.*, KASan [42]) or an oracle [98] to identify file system defects. This technique is easily adapted to new file systems and extensible with new operations, owing to the universality of syscalls. Nevertheless these implementations have lacked the versatility needed to explore diverse inputs and do not explore the state space like Metis. Furthermore, these testing methods typically identify only a limited range of bugs; for instance, CrashMonkey [98] exclusively detects crash-consistency bugs. We do not include a comparative analysis of testing for other storage systems, such as NVM libraries [35] and data structures [36], given their different testing targets and goals.

Ultimately, Metis is not designed to replace any existing technique; rather, we believe that it is an additional tool that offers a complementary combination of capabilities not found elsewhere.

Verified file systems For Metis, a reliable and ideally bug-free reference file system is critical. Verified file systems are built according to formally verified logic or specifications. For example, FSCQ [22] uses an extended Hoare logic to define a crash-safe specification and avoid crash-consistency bugs. Yggdrasil [121] constructs file systems that incorporate automated verification for crash correctness. DFSCQ [21] introduces a metadata-prefix specification to specify the properties of `fsync` and `fdatasync` for avoiding application-level bugs. SFSCQ [57] offers a machine-checked security proof for confidentiality and uses data non-interference to capture discretionary access control to preclude confidentiality bugs. However, the specifications of verified file systems have only been used to verify particular properties (*e.g.*, crash consistency [22, 121, 21] or concurrency [149]), so other unverified components can still contain bugs. Worse, even after rigorous verification, bugs can still hide due to erroneous specifications (*e.g.*, a crash-consistency bug reported on FSCQ [70]). None of these verified file systems include the extra APIs that RefFS provides, which are crucial for optimizing model-checking performance. While RefFS has not been formally verified, it relies on long-term Metis testing to attain high robustness. Thus, we chose it, rather than a verified file system, as the reference.

Chapter 4

IOCov: Input and Output Coverage for File System Testing

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.

4.1 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 [87, 147]. Various testing approaches have discovered such bugs and improved file system reliability [95, 14]. Testing file systems remains a challenge, however, due to their complexity, the presence of corner cases [142], their ongoing development [87], and the demand for strong resiliency (*e.g.*, crash consistency [108, 121]).

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 [87, 70, 69]. 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 [2], the most commonly used metric for evaluating test quality [44], 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 [58]. 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 [7, 2]; and (2) The code covered by tests may still hide bugs, depending on parameter values [59, 16].

To investigate the correlation between code coverage and testing effectiveness (*i.e.*, the ability to find bugs), we conducted a study (Section 4.2) 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 [37, 7]; 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 [118]—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* [50, 128, 74] 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* [5], 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 [87] (Section 4.2). 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 [127, 11]. 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 [87, 147].

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 [118] and CrashMonkey [98], and found many untested regions for both.

4.2 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 [51, 16, 102] or relatively simple user applications [59, 39]. 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 [91] and BtrFS [115]. First, because of the strong link between Git commits and *accepted* patches [61, 125], we manually analyzed the latest 100 Git commits from the Linux kernel repository [126] applied in 2022 for each file system—200 commits in total. Second, using Lu *et al.*'s technique [87], 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` [58] 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.

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 [89, 52]. 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 [12], less-tested inputs [79], and boundary values [129]; these are usually ignored by code-coverage metrics be-

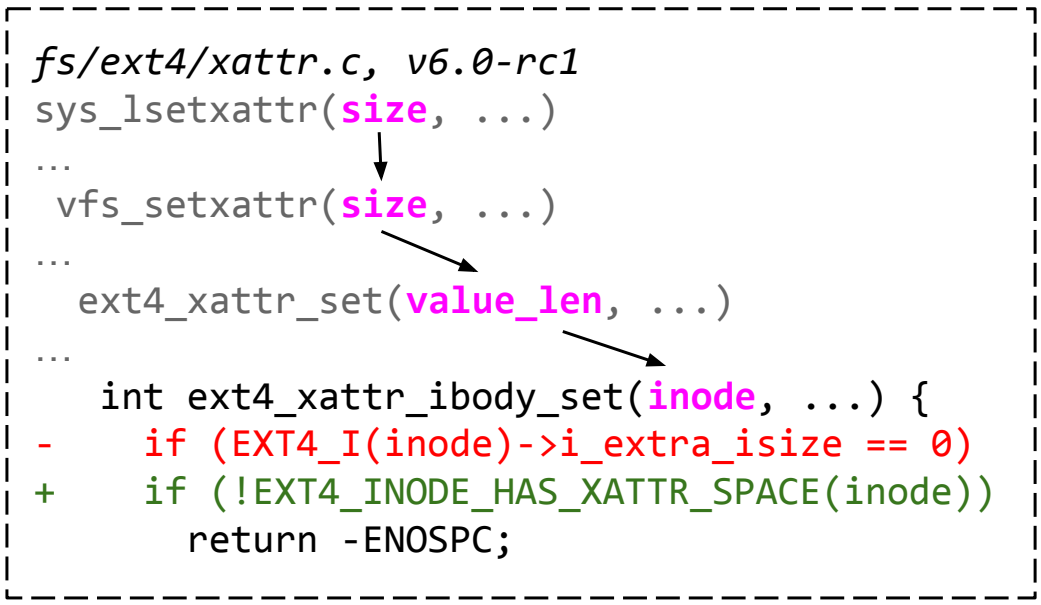


Figure 4.1: An example of a both input-related and output-related Ext4 bug. The bug was fixed by checking whether the inode has room to store additional xattrs in `ext4_xattr_ibody_set`.

cause they often execute the same code as heavily-tested inputs [127].

Figure 4.1 shows such an example from a recent bug [129] 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.

4.3 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 [127, 11]. 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 [113]. 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 [113, 105, 28, 148], but ultimately used powers of 2 as boundaries because they are common in file systems [64]. 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 [7].

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 [86, 4]. 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.

Many syscalls have variants with different prototypes (*e.g.*, `open`, `openat`, `creat`, and `openat2`). Variants share almost the same kernel implementation [127, 110], 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.

4.4 Evaluation

We experimented with the IOCov prototype on two file system testers: CrashMonkey [98] and xfstests [118]. 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.

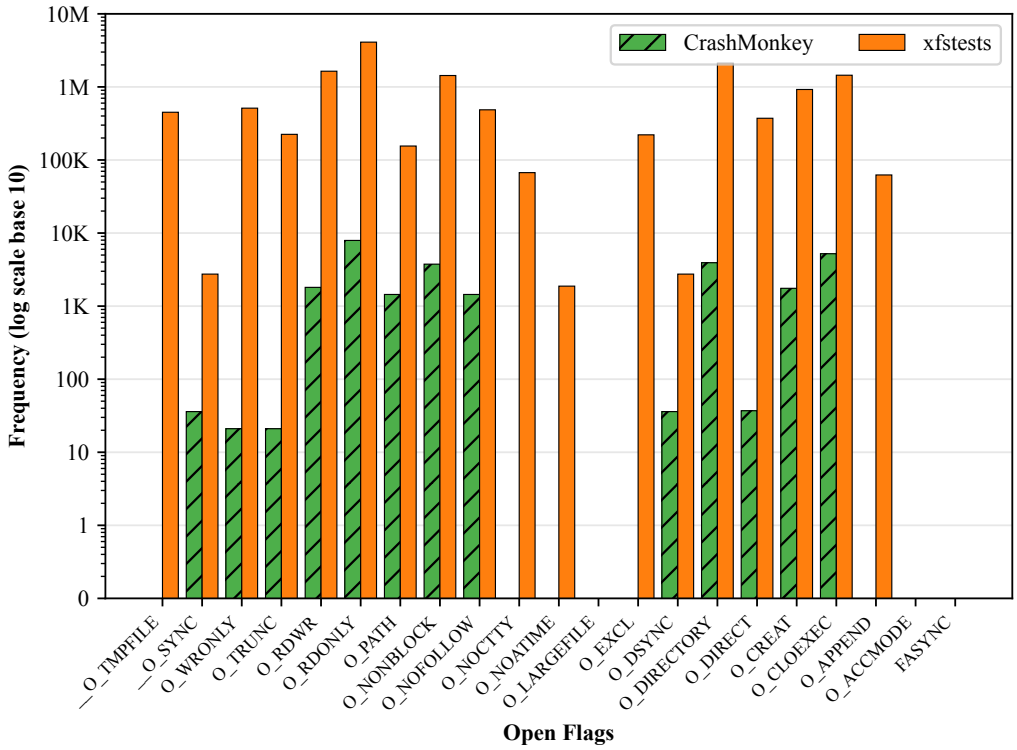


Figure 4.2: Input coverage of `open` flags for CrashMonkey and xfstests. The x-axis lists all possible flags supported by `open`. The y-axis (log_{10}) shows the frequency of each `open` flag exercised by each testing tool.

Input coverage results Figure 4.2 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 4.2 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 [133]). We also analyzed the number of

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 4.1: 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.

tested combinations of flags. Table 4.1 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 4.3 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 [6]) 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 [81]). The size 0 is also a boundary value because it is the minimum possible size accepted by `write` but is easily neglected by testing [113]. The y-axis (log_{10}) of Figure 4.3 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 4.3) despite the fact that 64-bit 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.

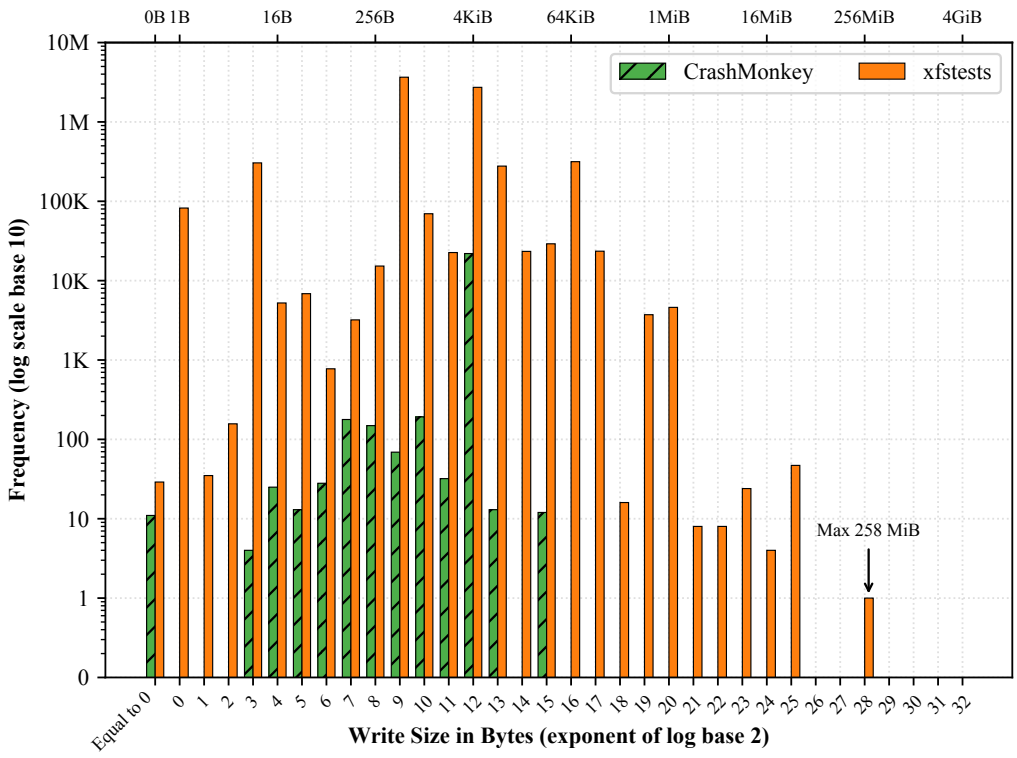


Figure 4.3: Input coverage of `write` size (in bytes) for CrashMonkey and xfstests. The x-axis shows the log_2 of write size, rounded down to the nearest boundary value. The $x2$-axis shows the actual size corresponding to the x-axis. For example, $x = 28$ represents 2^{28} or 256MiB. The y-axis (log_{10}) shows the frequency.

Output coverage results Figure 4.4 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*

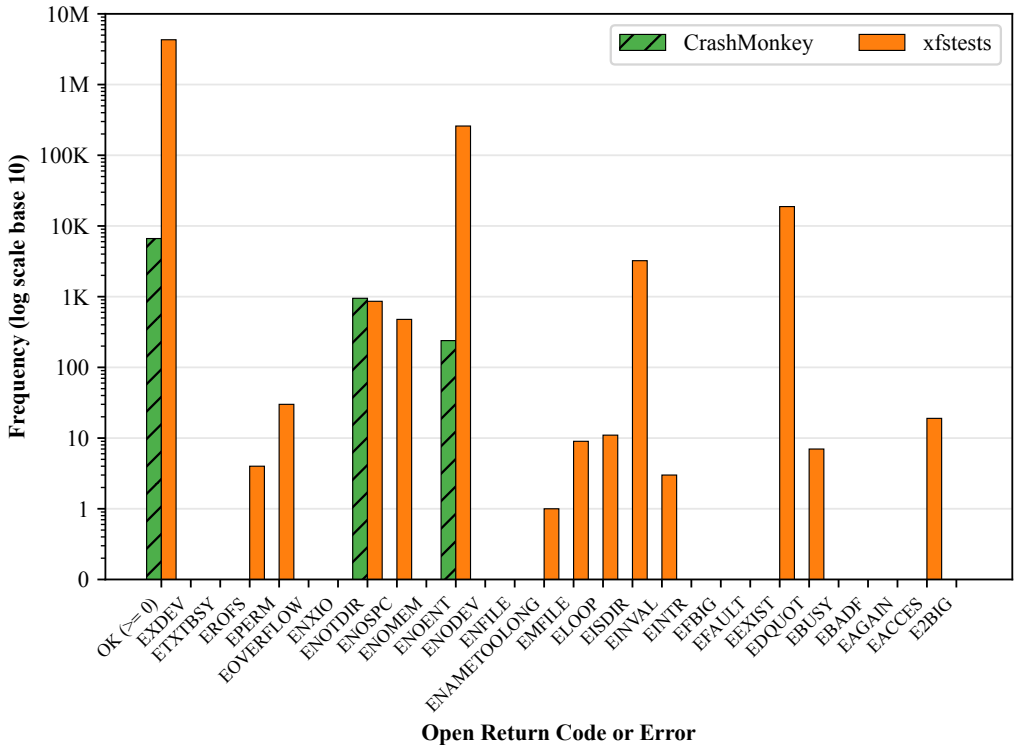


Figure 4.4: Output coverage of `open` for CrashMonkey and xfstests. The x-axis shows outputs (*i.e.*, success and error codes) returned by `open` and its variants. The y-axis (log_{10}) shows the frequency.

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

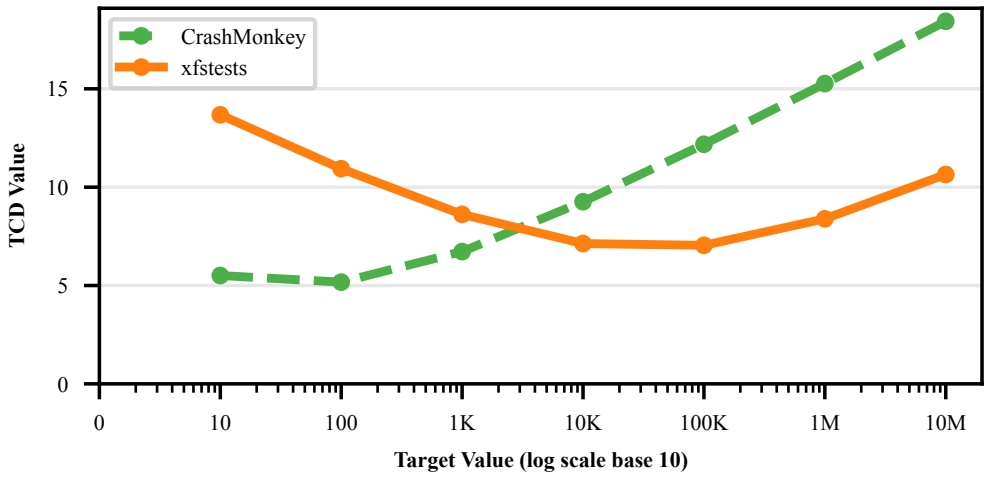


Figure 4.5: Test Coverage Deviation (TCD) for `open` flags, for CrashMonkey and xfstests. The x-axis (log_{10}) shows the target number of tests for each `open` flag. The y-axis shows the TCD value of each testing tool for different target values.

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 [66]):

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

We use logarithms for the frequencies and target because under-testing is more problematic than over-testing [80], 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 [98], 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 4.5 shows the TCD for CrashMonkey and xfstests with different

target arrays. The x-axis (log_{10}) shows the uniform value of the target array for `open` flags. We see that below $x \approx$5,237, CrashMonkey has a better (lower) TCD; above that value, xfstests is better. This matches Figure 4.2, 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.

4.5 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 [43, 140, 70, 138]. For different fuzzers, IOCov needs to apply other techniques to trace fuzzed syscalls. For example, Syzkaller [43] logs syscalls with declarative descriptions, which need to be parsed by IOCov. Hydra [140, 70], however, exercises syscalls with Library OS [109], 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.

Chapter 5

Metis: File System Model Checking via Versatile Input and State Exploration

We present *Metis*, a model-checking framework designed for versatile, thorough, yet configurable file system testing in the form of input and state exploration. It uses a nondeterministic loop and a weighting scheme to decide which system calls and their arguments to execute. Metis features a new *abstract state* representation for file-system states in support of efficient and effective state exploration. While exploring states, it compares the behavior of a file system under test against a reference file system and reports any discrepancies; it also provides support to investigate and reproduce any that are found. We also developed RefFS, a small, fast file system that serves as a reference, with special features designed to accelerate model checking and enhance bug reproducibility. Experimental results show that Metis can flexibly generate test inputs; also the rate at which it explores file-system states scales nearly linearly across multiple nodes. RefFS explores states 3–28× faster than other, more mature file systems. Metis aided the development of RefFS, reporting 11 bugs that we subsequently fixed. Metis further identified 15 bugs from seven other file systems, six of which were confirmed and with one fixed and integrated into Linux.

5.1 Introduction

File system testing is an essential technique for finding bugs [70] and enhancing overall system reliability [46], as file-system bugs can have severe consequences [87, 147]. Effective testing of file systems is challenging, however, due to their inherent complexity [7], including many corner cases [142], myriad functionalities [13], and consistency requirements (*e.g.*, crash consistency [108, 121]). Developers have created various testing technologies [141, 118, 98] for file systems, but new bugs (both in-kernel and non-kernel) continue to emerge on a regular basis [70, 140, 69].

To expose a file-system bug, a testing tool must execute a particular system call using specific inputs on a given file-system state [87, 142, 83]. For example, identifying a well-known Ext4 bug [77] requires a write operation on a file initialized with a 530-byte data segment. In this case, the write operation is an input, and the file with a specific size constitutes (part of) the file-system state. Recent work [83, 14] also underscored the importance of adequately covering both file-system inputs and states during testing. While existing testing technologies seek to cover a broad range of file systems' functionality, they often do not, however, integrate coverage of *both* file-system inputs and states [70, 140, 98, 20]. For example, handwritten regression tools like xfstests [118] can achieve good test coverage of specific file-system features [97, 7], but do not comprehensively cover syscall inputs; similarly, fuzzing techniques (*e.g.*, Syzkaller [43]) are designed to maximize code—not input—coverage [67].

Both the input and state spaces of file systems are too vast to be completely explored and tested [37, 18], so it is better to leverage finite resources by focusing on the most pertinent inputs and states [83, 143, 141]. For example, metadata-altering operations, such as `link` and `rename`, and states with a complex directory structure are more frequently utilized in POSIX-compliance testing [114]. Existing testing technologies also lack the versatility to test specific inputs and states [118, 98, 43]. Thus, new testing tools and techniques are needed [83, 87] to avoid under-testing (which could miss potential bugs) or over-testing (which wastes resources that may be better deployed elsewhere).

This paper presents Metis, a novel model-checking framework that enables thorough and versatile input and state space exploration of file systems. Metis runs two file systems concurrently: a file system under test and a reference file system to compare against [45]. Metis issues file-system operations (*i.e.*, system calls with arguments) as inputs to both file systems while simultaneously monitoring and exploring the state space via graph search (*e.g.*, depth-first search [53]).

To compare the relevant aspects of file-system states, we first abstract them and then compare the abstractions. The abstract states include file data, directory structure, and essential metadata; abstract states constitute the state space to be explored. Metis first nondeterministically selects an operation and then fills in syscall arguments through a user-specified weighting scheme. Next, it executes the same operation in both file systems and then compares both systems' abstract states. Any discrepancy is flagged as a potential bug. Metis evaluates the post-operation states to decide if a state has been previously explored; if so, it backtracks to a parent state and selects a new state to explore [53]. Metis continuously tests new file-system states until no additional unexplored states remain, logging all operations and visited states for subsequent analysis. Metis's replayer can reproduce potential bugs with minimum time and effort.

Metis effectively addresses the common challenges of model checking [25, 53] file systems. It checks file-system implementations directly, eliminating the need to build a formal model [101]. To manage large file-system input and state spaces, Metis enables parallel and distributed exploration [55] across multiple cores and machines. Metis works with any kernel or user file system, and does not require any specific utilities nor any modification or instrumentation of the kernel or the file system. It detects bugs by identifying behavioral discrepancies between two file systems without the need for oracles or external checkers, thus simplifying the process of applying Metis to new file systems. With few constraints, Metis is well suited for testing file systems that are challenging for other testing approaches, *e.g.*, file system fuzzing [70], that require kernel instrumentation and utilities. Nevertheless, the quality of the reference file system is pivotal for assessing the behavior of other file systems [45]. We therefore developed RefFS as Metis's reference file system. RefFS is an in-memory user-space POSIX file system with new APIs for efficient state checkpointing and restoration [123, 141]. Prior to using RefFS as our reference file system, we used Ext4 as the reference to check RefFS itself; Metis identified 11 RefFS bugs that we fixed during that process. Subsequently, we deployed 18 distributed Metis instances to compare RefFS and Ext4 for one month, totaling 557 compute days across all instances and executing over 3 billion file-system operations without detecting any discrepancy. This ensured that RefFS is robust enough to serve as Metis's (fast) reference file system.

Our experiments show that Metis can configure inputs more flexibly and cover more diverse inputs compared to other file-system testing tools [118, 98, 43]. Metis's exploration rate scales nearly linearly with the number of Metis instances, also known as verification tasks (VTs). Despite being a user-level file system,

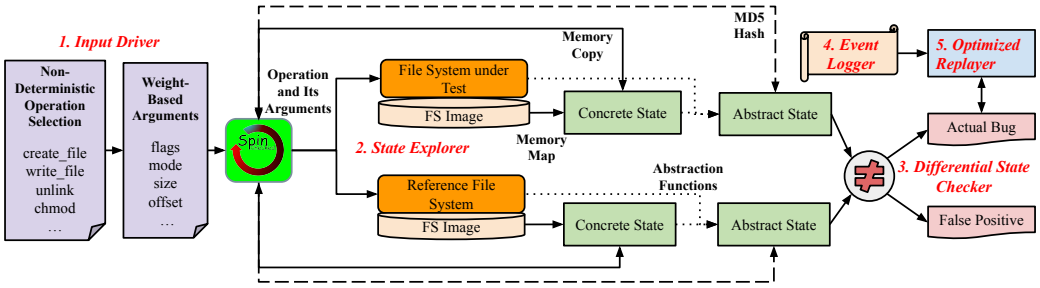


Figure 5.1: Metis architecture and components. From left to right, Metis generates syscalls and their arguments that are executed by both file systems, determines resulting states, and checks for discrepancies between states. The Logger records all the operations for convenient bug replay by the Replayer. The SPIN model checker stores previous state information for state exploration.

RefFS's states can be explored by Metis 3–28× faster than other popular in-kernel file systems (*e.g.*, Ext4, XFS, Btrfs). Using Metis and RefFS, we discovered 15 potential bugs across seven file systems. Of these, 13 were confirmed as previously unknown bugs, six of which were confirmed by developers as real bugs. Moreover, one of those bugs—which the developers confirmed existed for 16 years—and the fix we provided, was recently integrated into mainline Linux.

In sum, this paper makes the following contributions:

1. We designed and implemented Metis, a model-checking framework for versatile and thorough file-system input and state-space exploration.
2. We designed and implemented an effective abstract state representation for file systems and a corresponding differential state checker.
3. We designed and implemented the RefFS reference file system with novel APIs that accelerate and simplify the model-checking process.
4. Using RefFS, we evaluated Metis's input and state coverage, scalability, and performance. Our results show that Metis, together with RefFS, not only facilitates file-system development but also effectively identifies bugs in existing file systems.

5.2 Background and Motivation

In this section, we first introduce the procedures and challenges for testing and model-checking file systems. We then discuss two vital dimensions for file system

testing: input and state. We demonstrate the challenges of achieving versatile and comprehensive coverage of both inputs and states.

File system testing and model checking File systems can be tested statically or dynamically. Static analysis [95, 14] evaluates the file system's code without running it; while useful, it struggles with complex execution paths that may depend on runtime state. Our work therefore emphasizes dynamic testing—executing and checking file systems in real-time scenarios [114, 20, 98]. Generally, dynamic testing involves (1) crafting test cases using system calls, (2) initializing the file system, (3) running the test cases, and (4) post-execution validation of file system properties. Hence, the quality of test cases directly affects the testing efficacy.

Model checking is a formal verification technique that seeks to determine whether a system satisfies certain properties [25, 130]. The model is typically a state machine, and the properties, usually expressed in temporal logic, are checked using state-space exploration [24]; here, each state represents a snapshot of the system under investigation. To automate this process, model checkers (such as SPIN [53]) are used to generate the state space, verify property adherence, and provide a counterexample when a property is violated.

Extracting a model from a system implementation can be challenging, especially for large systems like file systems [142, 141]. Thus, recent work on implementation-level model checking [142, 141] seeks to check the implementation directly (without a model). Such approaches [141] require one to create new, specialized checkers to test new file systems, and these checkers are typically focused on a limited range of bugs, such as crash-consistency bugs [142, 141]. The ongoing challenge is to simplify implementation-level file-system model checking so that using it does not require extensive effort or significant expertise in model checking and file systems, while at the same time being able to identify a wide range of bugs.

Covering system calls and their inputs We refer to the system calls (syscalls) and their arguments as *inputs* or *test inputs* because syscalls are commonly used by user-space applications—and thus testing tools—to interact with file systems [38, 135]. Thoroughly testing file system inputs is challenging. While file-system–related syscalls represent only a subset of all Linux syscalls [127, 11], each syscall has multiple arguments, and the potential value range for these arguments is vast [127, 83]. For example, `open` returns a file descriptor, accepting user-defined arguments for `flags` and `mode` in addition to `pathname`. Both `flags` and `mode`

are bitmaps with 23 and 17 bits, respectively, representing many possible combinations. The bits represented in `flags` alone have 2^{23} possible values, leading to an aggregate input space of 2^{40}. Similarly, `write` and `lseek` take 64-bit-long byte-count arguments that have a large input domain of 2^{64} possible values. Nevertheless, it is vital to test as many representative syscall inputs as possible.

Fully testing all syscalls with every potential argument is impractical [43, 63]. Instead, a sensible approach [74, 83] is to *segment* a large input space into multiple, disjoint input partitions—called *input space partitioning* [132, 83, 65]. How much a testing tool examines input partitions is called *input coverage* [50, 128, 74]. Utilizing input partitions and coverage, testing tools can target the coverage of different partitions—each representing a subset of analogous test inputs. Intuitively, file system developers recognize the need to, say, separately test critical I/O write sizes of 512 and 4096; conversely, once one tests an I/O size of, say, 5000 bytes, the gains from testing subsequent adjacent sizes (*e.g.*, 5001, 5002, . . .) quickly diminish.

To compute input coverage, we categorized each syscall's arguments into four classes [83, 11, 127]: (i) identifiers (*e.g.*, file descriptors), (ii) bitmaps (*e.g.*, `open` flags), (iii) numeric arguments (*e.g.*, `write` size), and (iv) categorical arguments (*e.g.*, `lseek` "whence"). We partitioned the input space using type-specific methods. For example, bitmaps are partitioned by each flag and certain combinations thereof. Numeric arguments are partitioned by boundary values (*e.g.*, powers of 2 [64]). Our goal is to achieve thorough input coverage while configuring it based on test strategies to customize the overall search space. To the best of our knowledge, no existing file system testing method is specifically designed for comprehensive input coverage, nor are there any techniques to flexibly define the input's coverage.

Challenges of testing file system states In file system testing, the *state* refers to the content, status, and full context of the file system at a given point in time [123, 37]. Comprehensive state exploration is important as certain bugs manifest exclusively under specific states [87, 77, 129]. Numerous file system states can be explored when some existing testing approaches [118, 98] execute operations. Yet the majority of these approaches lack state tracking—the ability to record and identify previously or similarly visited states—thus wasting resources [141]. The challenges are thus twofold: state definition and efficient state tracking.

Defining file system states involves a tradeoff, because components such as on-disk content, in-memory data, configuration, kernel context, and device types

are all candidates for inclusion in the state [37]. An overly detailed state definition can render state exploration infeasible due to resources spent on visiting multiple states that should be treated as if they were identical [25]. Conversely, an overly narrow definition can skip key states and potentially miss defects [19]. Therefore, one should be able to define the state space flexibly, so it contains all desired file system attributes while maintaining a manageable state space.

Due to massive state spaces, state tracking incurs considerable overhead, thus slowing the entire exploration process. While model checkers provide a mechanism for state exploration [53] with state tracking and certain optimizations, they still have to contend with the state explosion problem—a significant challenge where the number of system states grows exponentially with the number of system variables, making state exploration computationally impractical [25]. In file systems, this issue is exacerbated by the inherently slow nature of I/O. An alternative approach is to partition the state-exploration process across multiple instances, with each instance exploring a certain portion of the state space; doing so requires a sophisticated design for diversified, parallel exploration [55].

5.3 Design

In this section, we describe Metis's design principles and operation. We explain how Metis meets the challenges of exploring file system inputs and states, and how it provides versatility.

Metis architecture As shown in Figure 5.1, Metis has five main components: (1) Input Driver, (2) State Explorer, (3) Differential State Checker, (4) Event Logger, and (5) Optimized Replayer. Each component is designed to be independent, allowing for modularity and extensibility.

The Input Driver (§5.3.1) generates syscalls and arguments to serve as the test inputs to both file systems. Metis is built on top of the SPIN model checker [53] to combine input selection with state exploration. The State Explorer (§5.3.2) extracts concrete and abstract states from both file systems and interfaces with SPIN to explore new states. The Differential State Checker (§5.3.3) verifies that both file systems have identical behavior after each operation, by comparing their abstract states, syscall return values, and error codes. Any discrepancies are reported by the checker and treated as potential bugs. The Event Logger and the Optimized Replayer (§5.3.4) help analyze reported discrepancies and reproduce potential bugs more efficiently.

5.3.1 Input Driver

Metis's Input Driver maintains a list of operations from which the SPIN model checker can repeatedly and nondeterministically choose what to execute, including individual syscalls (*e.g.*, `unlink`) as well as meta-operations comprising a (small) sequence of syscalls (*e.g.*, the `write_file` operation opens a file and writes to it at a specific offset). From a given file system state, multiple potential successor states may arise. Through its nondeterministic choices of operations, Metis can effectively explore many of these options, ensuring thorough state exploration. To bound the input space, each operation randomly picks a file or directory name from a predetermined set of pathnames. The Input Driver is flexible and can generate files or directories with arbitrarily deep directory structures, long pathnames, and other unexpected scenarios such as many files inside a single directory.

We focus on state-changing operations [45] (*i.e.*, not read-only ones) as the Input Driver seeks to maximize the exploration of file system states. Currently, the Input Driver supports five meta-operations (`create_file`, `write_file`, `chown_file`, `chgrp_file`, and `fallocate_file`), and 10 individual syscalls (`truncate`, `unlink`, `mkdir`, `rmdir`, `chmod`, `setxattr`, `removexattr`, `rename`, `link`, and `symlink`). Adding a new operation has minimal effort of about 10 LoC. Metis exercises read-only operations such as `read`, `getxattr`, and `stat` after each state-changing operation, when computing file system abstract states in the State Explorer (§5.3.2).

After selecting the operation, Metis chooses its arguments based on a series of user-specified weights that control how often various argument partitions (§5.2) are tested. In the Input Driver, weights represent the probabilities assigned to different input partitions, which control testing frequencies. The method of assigning weights varies based on the argument type [11, 83]. For bitmap arguments, each bit receives a probability of being set. The number of input partitions in a bitmap argument is equivalent to its individual bit count. Given the ubiquity of powers of 2 in file systems [64], numeric arguments like `write` size (requested byte count) have input partitions segmented by these numbers as boundary values, rounding down to the nearest boundary. For example, write sizes ranging from 1024 to 2047 bytes (2^{10} to $2^{11} - 1$) are grouped in the same partition. Assigning a weight (*e.g.*, 15%) to this partition implies a 15% chance of selecting a write size between 1024 and 2047 bytes. The total weight of all write-size partitions equals 100%. We placed 0 bytes as a distinct partition (unusual but allowed under POSIX) because the smallest power of 2 is 1, which is greater than 0. Additionally, Metis

can also be configured to test only boundary values (powers of 2) such as 4096 as well as near-boundary values (± 1 from the boundary, *e.g.*, 4095/4097) that are useful for testing underflow and overflow conditions.

The choice of weights depends on the user's objectives. For example, while `O_SYNC` is common in crash-consistency testing [98], it is used infrequently for POSIX compliance [114]. Due to disk I/O's slow speed, many tests focus on small write sizes [20]. However, testing larger sizes can uncover size-specific bugs [129, 114]. Our objective is to ensure that Metis remains versatile and to allow one to adjust the input weights in line with the test focus.

5.3.2 State Exploration and Tracking

Problem	**Cause of discrepancies**	**Solution**
Different directory size for same contents	Size calculation methods	Ignore directory sizes
Different orders of directory entries	Internal data structures	Sort the output of `getdents`
FS-specific special files and directories	Internal implementations	Create an exception list of special entries
Different usable data capacities	Space reservation and utilization	Equalize free space among file systems

Table 5.1: Examples of false positives identified and addressed by Metis.

State explorer The objective of Metis's State Explorer is to use graph traversal to conduct thorough and effective "state graph exploration," where the nodes correspond to file-system states and the edges represent transitions caused by operations [24]. Metis supports depth-first search (DFS) as the main search algorithm.

The State Explorer relies on the SPIN model checker [53] to conduct the state-space exploration. SPIN supports the Promela model-description language, and allows embedding C code in Promela code. This capability allows us to seamlessly issue low-level file-system syscalls and invoke utilities. SPIN's role is to provide optimized state-exploration algorithms (*e.g.*, DFS) and data structures to track and store the status of the state graph; thus, we do not have to implement these features in the State Explorer.

In model checking, there are two types of states: *concrete* and *abstract*. Concrete states contain all the information that describes the states of the file system

being checked. Abstract states serve as signatures to identify different system states of interest during the exploration.

After each operation, the State Explorer calls the abstraction function to extract abstract states as hash values from both file systems. Every time an abstract state is created, SPIN checks whether it has already been visited by looking up the abstract state in SPIN's hash table and decides on the next action, either backtracking to a previous concrete state or continuing from the current one. Meanwhile, the State Explorer `mmap`s the full file-system image into memory to be tracked by SPIN as a concrete state. Concrete states are stored in SPIN's stack to allow the State Explorer to restore the full file-system state as required. To improve the performance of state exploration, we use RAM disks as backend devices for on-disk file systems. In Metis, we create both file systems with the minimum device sizes to reduce the memory consumption of maintaining concrete states and to make it easier to trigger corner cases such as `ENOSPC`.

File system abstract states A *concrete state* is a reflection or snapshot of the entire (and highly detailed) file-system image, which renders it inappropriate for distinguishing a previously visited state [19]. This is because any small change to the file-system image leads to a new concrete state, even though there may be no "logical" change in the file system. For example, Ext4 updates timestamps in the superblock during each mutating operation, even if no actual change to a user-visible file was made. This substantially expands the state space, with many states differing only by minor timestamp changes, and leads to wasted resources on logically identical states. Additionally, because file systems are designed with different physical on-disk layouts, we cannot use concrete states to compare their behaviors. Therefore, we need a different state representation that includes only the essential and comparable attributes common to both file systems.

To address this problem, we defined an *abstraction function* to calculate file-system *abstract states* to distinguish unique states, and to compare file system behaviors. The abstract state contains pathnames, data, directory structure, and important metadata for all files and directories (*e.g.*, mode, size, nlink, UID, and GID); we exclude any noisy attributes such as `atime` timestamps. We then hash this information to compact the abstract state for a more effective comparison. Metis supports several hash functions to compute abstract states; we evaluated the speed and collision resistance of each hash function (results elided for brevity) and chose MD5 by default as it had the best tradeoff of those characteristics.

The abstraction function deterministically aggregates key file system data and

metadata, enabling comparison across different file systems. Specifically, the abstraction function begins by enumerating all files and directories in the file system by traversing it from the mount point. Their pathnames are sorted into a consistent, comparable order. We then `read` each file's contents and call `stat` to extract its important metadata mentioned above, following the pathname order. Finally, we compute the (MD5) hash based on the files' content, directory structure, important metadata, and pathnames to acquire the abstract state. Using abstract states not only prevents visiting duplicate states but also significantly reduces the amount of memory needed to track previously-visited states, owing to our lightweight hash representation, which in turn boosts Metis's exploration speed.

Tracking full file system states In addition to abstract states, another complexity in tracking file system states is saving and restoring the concrete states when Metis needs to backtrack to a previous state (*i.e.*, when reaching an already visited state); this involves State Save/Restore (SS/R) operations for concrete states. Concrete states must contain all file system information including persistent (on-disk) and dynamic (in-memory) states. Metis can feasibly save and restore on-disk states by copying the on-disk device and subsequently copying it back. Kernel file systems (*e.g.*, Ext4 [91]) maintain states in kernel space, which is inaccessible to Metis, a user process. Similarly, user-space file systems built on libFUSE (*e.g.*, fuse-ext2 [3]) are separate processes with separate address spaces, so again Metis cannot directly track their internal state. Tracking only persistent on-disk state leads to cache incoherency, because cached in-kernel information is inconsistent with the on-disk content.

We tried and evaluated several approaches to tracking full file system states (performance results elided for brevity) including `fsync` syscall, `sync` mount option, process snapshotting [26, 139], VM snapshotting [71, 75], and LightVM [90]. None of these approaches were effective due to their functional deficiencies or inefficient performance. For those reasons, we adopted the approach presented in [123] to unmount and remount the file system between *each* operation in Metis. An unmount is the *only* way to fully guarantee that no state remains in kernel memory. Remounting guarantees loading the latest on-disk state, ensuring cache coherency between each state exploration. This unmount-remount method was a compromise that ensures data coherency yet provides reasonable performance (§5.6.2), especially coupled with our specialized RefFS (§5.4).

5.3.3 Differential State Checker

Metis checker goals and approaches Using only the Input Driver and State Explorer would constrain the detection of bugs to those manifesting as visible symptoms [20], such as kernel crashes. We thus needed a dedicated checker to identify cases where file systems fail silently [70] (*e.g.*, data corruption). Moreover, existing checkers usually require considerable effort to be applied to newly developed or constantly-evolving file systems. For example, since many checkers are hand-written (*e.g.*, xfstests), the testing of new file systems involves redesigning and refactoring test cases. Some checkers depend on an exact (*e.g.*, POSIX) specification or an oracle for bug detection [98, 114]: they are difficult to adapt to continuously-evolving file systems.

File systems vary considerably in terms of their developmental stages [145, 87]: mature file systems are typically more stable than new, emerging, or less popular ones [87]. Yet many still share common (POSIX) features and data-integrity requirements. Therefore, we rely on a *differential testing* approach [92], to check emerging file systems for silent bugs, eliminating the need for a detailed specification or an oracle.

We developed Metis's Differential State Checker to identify a broad range of file system bugs and facilitate file system development. Our checker can easily adapt to test new file systems; it requires no modification to the checker, only a replacement of the file system under test. Metis uses a well-tested, reliable file system as the reference file system and a less-tested, emerging one as the file system under test. After each file system operation, the Differential State Checker compares the resulting states of both file systems to detect any discrepancies. To prevent false positives, it only compares the common attributes of file systems, including their abstract states, return values, and error codes.

Eliminating false positives As any discrepancy is reported as a potential bug, when developing Metis we found that it sometimes identified discrepancies that were not bugs (*i.e.*, false positives). We implemented measures to avoid these false positives. Table 5.1 summarizes several such cases including their problems, causes, and solutions.

All these discrepancies arose due to different file system designs and implementations. For instance, Ext4 has a special `lost+found` directory and computes directory sizes by a multiple of the block size. In contrast, other file systems report sizes by the number of active entries and do not have a `lost+found` directory. Despite the same device sizes for different file systems, the available space varies

due to different utilized and reserved space (*e.g.*, for metadata). To address this, we equalize free space among file systems by creating dummy files based on the differences in their available spaces.

While developing Metis, we analyzed every discrepancy we encountered and addressed all false positives. Whenever a false positive was identified, we updated the state abstraction function or file system initialization code to eliminate such instances, an infrequent process that was conducted manually. None of these solutions introduce false negatives, because they all deal with non-standardized behavior. For example, an application should not expect sorted output from `getdents`. Nevertheless, if a change introduces any misbehavior, Metis's Differential State Checker will report and handle it.

5.3.4 Logging and Bug Replay

When detecting a discrepancy, it is important to be able to analyze the operations executed by the file systems to identify and reproduce the potential bug. Thus, Metis's Event Logger records details of all file-system operations and outcomes, comprising every syscall and their arguments, return values, error codes, SS/R operations, and resultant abstract state. Additionally, the Event Logger logs file-system information such as the directory structure and important metadata to pinpoint the deviant behavior as soon as a discrepancy is detected. To reduce disk I/O, we store the runtime logs in an in-memory queue and periodically commit them to disk. Leveraging the Event Logger, we can reproduce the precise sequence of operations leading to a discrepancy found by Metis.

Metis can replay identified bugs by re-executing the operations from the start of Metis's run. This process can be time-consuming, however, if the discrepancy was detected after executing many operations and passing through numerous states [4]. So we needed a way to reproduce a discrepancy quickly. Existing test-case minimization techniques [146, 70] remove one operation from a sequence until the remaining operations can reproduce the bug; but this trial-and-error process is slow due to the abundance of I/O operations.

To replay bugs efficiently, the Optimized Replayer reproduces them using only a few operations (recorded in logs) and one (concrete state) file system image. Using SPIN, we retain concrete states in a stack, thereby capturing all file-system images along the current exploration path and allowing for bug reproduction from any desired location in the stack. Recent findings [98, 70] indicate that most bugs can be reproduced on a newly created file system using a sequence of eight or fewer operations. Accordingly, Metis uses an in-memory circular buffer to

retain pointers to a few of the most recent file-system images (defaults to 10, but configurable) for quick post-bug processing. In practice, we first attempt to reproduce the bug using the most recent image (immediately preceding the bug state) along with the latest operation. If unsuccessful, we turn to the previous image and the two last operations, and so on in a similar pattern. This eliminates the need for Metis to replay the entire operation sequence from the beginning.

5.3.5 Distributed State Exploration

Along with performing state abstraction and setting limits on the number of files and directories, we also restrict the search depth to control the exponential growth of the state space. We set the maximum search depth to 10,000 by default [53]. If the search hits the $10{,}000^{th}$ level, Metis reverts to the prior state rather than exploring deeper. Thus, the state space becomes bounded, allowing Metis to perform an exhaustive search. Still, even with this depth restriction, the state space remains large because of the variety in test inputs and file system properties [37]. Exploring this space using a single Metis process (called a *verification task*, or VT) requires significant time.

To parallelize the state-space exploration [54] we use Swarm verification [55], which generates parallel VTs based on the number of CPU cores. Each VT examines a specific portion of the state space. To prevent different VTs from re-exploring the same states, and to avoid having to coordinate states across VTs, SPIN employs several *diversification* techniques [55], where every VT receives a unique combination of bit-state hash polynomials, number of hash functions, random-number seeds, search orders (*e.g.*, forward or in reverse) and search algorithms (*e.g.*, DFS), ensuring varied exploration paths.

We enabled these parallel and distributed exploration capabilities for Metis. The setup uses a configuration file to determine the machine and CPU core count; Metis then produces the exact VT count based on the configuration file. When Metis runs on distributed machines, each runs a handful of VTs, one per CPU core. Each VT is automatically configured with a distinct combination of diversification parameters, guiding them to explore different state space areas. Utilizing multiple Metis VTs across multiple cores and machines increases the overall speed of state exploration while testing more inputs. Every Metis VT operates independently, with its own device, mount point, and logs, without interference with other VTs. Given that VTs explore states autonomously without inter-VT communication, there is a risk of resource wastage if several VTs examine the same state [55]. We deployed multiple VTs on several multi-core machines and evaluated Metis

extensively under Swarm verification (§5.6.2).

5.3.6 Implementation Details

Metis uses SPIN to achieve basic model-checking functions. The Promela modeling language [53] serves as the main interface with SPIN. We wrote 413 lines of Promela, consisting of `do...od` loops that repeatedly select one of a number of cases in a nondeterministic fashion. Each case issues file-system operations, performs differential checks, and records logs. The main part of Metis comprises 7,911 lines of C/C++ code that implement Metis's components and its communication with SPIN. We also created 1,230 lines of Python/Bash scripts to manage different Metis VTs and runtime setup, such as invoking `mkfs`, and creating mount points and devices. We created RAM block devices as backend storage for on-disk file systems. Linux's RAM block device driver (`brd`) requires all RAM disks to be the same size. We modified it (renamed `brd2`), to allow different-sized disks for file systems with different minimum-size requirements. We used `brd2` to create devices for on-disk file systems during the evaluation.

We changed 72 lines of SPIN's code (Aug 2020 version) to add dedicated hook functions for file system SS/R operations. Lastly, we added 31 lines of code to the original Swarm verification tool (Mar 2019 version) to enable more flexible compilation options and smoother compatibility with Metis.

In our experience, adding a new file system operation to Metis is straightforward. It requires only one additional case in the Promela code, amounting to about 10 lines. Most functionality in Metis is file-system-agnostic, *e.g.*, deploying the file system and computing abstract state. To test a new file system, we need to specify only the device type (*e.g.*, RAM disk for most file systems, MTD block device for JFFS2) and the desired device size in Metis.

5.3.7 Limitations of Metis

False negatives Like many other tools, Metis might experience false negatives: it could fail to detect an existing bug. First, since Metis's abstract state excludes time-related attributes, it cannot detect, *e.g.*, `atime`-related bugs. Though that is an unavoidable consequence of abstraction, we strive to make the abstract state as comprehensive as possible. Second, Metis identifies bugs by detecting behavioral discrepancies between the reference file system and the file system under test. Given the nature of differential testing [45, 92], Metis could fail to detect bugs shared between both file systems as no discrepancy would be found. To

address this problem, one can either use a flawless reference file system or leverage N-version programming [9], comparing more than two file systems, to reduce the probability that the same bug is present across all of them. Unfortunately, a completely bug-free file system does not exist. Despite recent efforts to formally verify certain file system properties, these verified file systems may still hide bugs [22]. Furthermore, while Metis was programmed to test any number of file systems concurrently, employing a majority voting scheme on more than two adds overhead and slows exploration. (That is one reason why we support distributed verification: to increase the overall exploration rate.)

Test overhead As Metis tracks both abstract and concrete states, it inevitably introduces extra overhead due to memory demands and the time taken for comparisons. Metis retains file system images in memory for state backtracking, although we limited memory consumption to the extent possible by choosing a minimum device size and restricting search depth. For file systems with a relatively small device-size requirement, such as Ext4 (256KiB minimum), Metis's peak memory consumption remains relatively low (2.4GiB). However, a file system with a larger minimum device size inherently consumes more memory. For example, XFS has a minimum size of 16MiB, leading to a potential memory use of 156GiB when we use a maximum depth of 10,000. To mitigate this issue, we reduced SPIN's maximum search depth below the default 10,000, decreasing resource and memory consumption while concomitantly reducing the size of the state space. Although we experimented with memory compression (*i.e.*, `zram` [48]) and added swap space to increase effective memory capacity, these choices actually reduced the overall state-exploration rate. The necessity of mounting and unmounting between each operation introduces additional time overhead to Metis. Since doing so is necessary for tracking full file system states, we mitigated this cost by deploying more VTs on multiple machines and using RAM disks.

Bug detection and root-cause analysis At present, Metis lacks the capability to identify crash-consistency and concurrency bugs in file systems. Due to the absence of crash state emulation [98, 76], Metis cannot find bugs that arise solely during system crashes. We plan to provide the option of invoking utilities such as `fsck` [107] between each Metis unmount/mount pair to help detect crash-consistency bugs. Given that Metis operates on file systems from a single thread, it tends to miss concurrency bugs (*e.g.*, race conditions [138]). While Metis's replayer assists in reproducing bugs, another limitation is Metis's inabil-

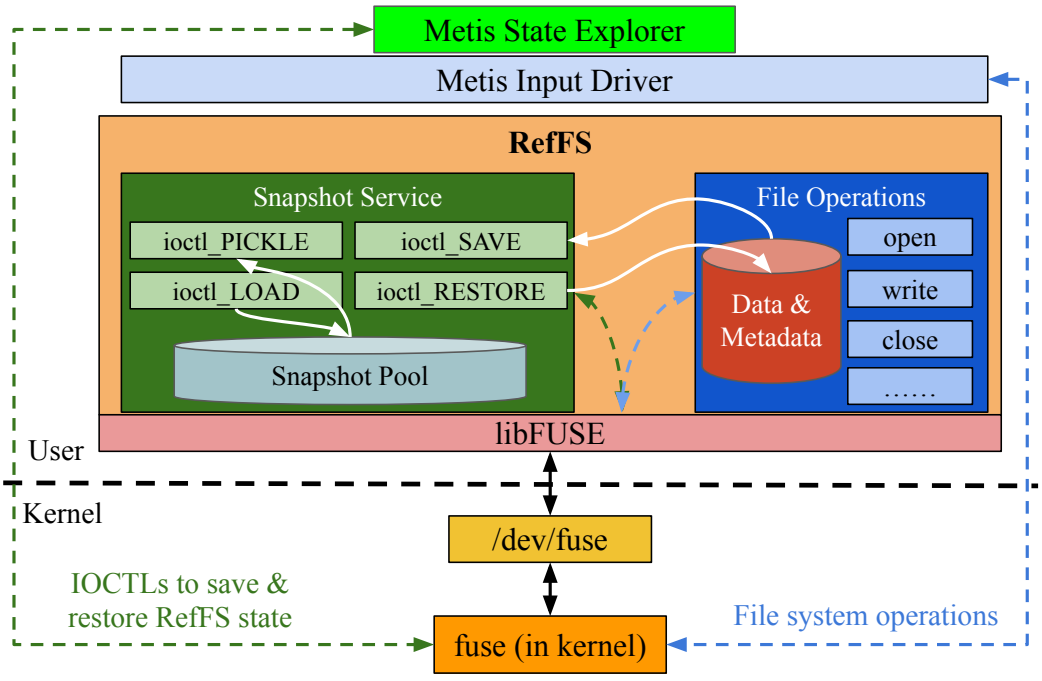


Figure 5.2: RefFS architecture and its interaction with Metis and kernel space. RefFS supports standard POSIX operations and provides snapshot services with a snapshot pool and four new APIs.

ity to precisely identify the root cause of detected state discrepancies within the code [116].

5.4 RefFS: The Reference File System

In Metis, the reference file system must reliably represent correct behaviors and ensure efficiency in the file system and SS/R operations. We initially chose Ext4 as the reference file system due to its long-standing use and known robustness [91]. Still, no file system, including Ext4, is absolutely bug-free. Additionally, Ext4 lacks optimizations for model-checking state operations, limiting its suitability. We believe that a reference file system should be lightweight [121, 22], easily testable and extensible, robust, and optimized for SS/R operations in model checking. Originally, we tried to modify small in-kernel file systems (*e.g.*, ramfs), to track their own state changes. However, capturing and restoring their entire

state proved extremely challenging because the state resides across many kernel-resident data structures [8]. Consequently, we developed a new file system, called RefFS, specifically designed to function as the reference system.

RefFS architecture RefFS is a RAM-based FUSE file system. Figure 5.2 shows the architecture of RefFS and its interplay with Metis and relevant kernel components. It incorporates all the standard POSIX operations supported by the Input Driver along with the essential data structures for files, directories, links, and metadata. We developed RefFS in user space to avoid complex kernel interactions and have full control over its internal states. Comprising 3,993 lines of C++ code, RefFS uses the `libFUSE` user-space library together with `/dev/fuse` to bridge user-space implementations and the lower-level `fuse` kernel module. Metis handles file system operations on RefFS in the same manner as other in-kernel file systems. Most importantly, RefFS also provides four novel snapshot APIs to manage the full RefFS file system state via `ioctl`s: `ioctl_SAVE`, `ioctl_RESTORE`, `ioctl_PICKLE`, and `ioctl_LOAD`. These are described next.

5.4.1 RefFS Snapshot APIs

RefFS shows how file systems *themselves* can support SS/R operations in model checking through snapshot APIs. The essence of SS/R operations lies in their ability to save, retrieve, and restore the concrete state of the file system. Although RefFS is an in-memory file system lacking persistence, it possesses a concrete state (*i.e.*, snapshot) that includes all information associated with the file system. Existing file systems like BtrFS [115] and ZFS [13], which support snapshots, can only clone (some of) the persistent state but not their in-memory states. In contrast, RefFS can capture and restore the in-memory states through its own APIs. Since RefFS stores all its data in memory, it guarantees saving and restoring the entire file system state.

Snapshot pool The snapshot pool is a hash table that organizes all of RefFS's snapshots; the key is the current position in the search tree. The value associated with each key is a snapshot structure that saves the full file system state including all data and metadata such as the superblock, inode table, file contents, directory structures, etc. The memory overhead of the snapshot pool is low because the size of the pool is smaller than Metis's maximum search depth. Because RefFS is a simple file system, the average memory footprint for each state is just 12.5KB.

Save/Restore APIs The `ioctl_SAVE` API causes RefFS to take a snapshot of the full RefFS state and add an entry to the snapshot pool. The `ioctl_RESTORE` does the reverse, restoring an existing snapshot from the pool. When Metis calls `ioctl_SAVE` with a 64-bit key, RefFS locks itself, copies all the data and metadata into the snapshot pool under that key, and then releases the lock. Similarly, `ioctl_RESTORE` causes RefFS to query the snapshot pool for the given key. If it is found, RefFS locks the file system, restores its full state, notifies the kernel to invalidate caches, unlocks the file system, and then discards the snapshot.

Pickle/Load APIs Unlike other file systems, RefFS maintains concrete states by itself in the snapshot pool, so Metis does not need to keep RefFS's concrete states in its stack. To ensure good performance, RefFS's snapshot pool resides in memory. However, this means that all snapshots are lost when RefFS is unmounted, which would make it challenging to analyze and debug RefFS from a desired state. Thus, committing these snapshots to disk before Metis terminates is important to ensure they are available for post-testing analysis and debugging. Given a hash key, the `ioctl_PICKLE` API writes the corresponding RefFS state to a disk file. It can also archive the entire snapshot pool to disk. Likewise, the `ioctl_LOAD` API retrieves a snapshot from disk, loading it back into RefFS to reinstate the file system state. Using the `ioctl_PICKLE` and `ioctl_LOAD` APIs, RefFS can flexibly serialize and revert to any file system state both during and after model checking, aiding bug detection and correction. Specifically, these APIs allow RefFS to gain the same benefits as Metis's post-bug replay and processing, enabling bug reproduction from any point in a Metis run.

5.5 The Case of Checking Distributed File Systems

In this section, we outline the structure and procedure for checking the NFS kernel server and NFS-Ganesha using Metis, as well as the benefits of using RefFS as both the NFS local and reference file systems.

5.5.1 The Architecture of Checking NFS

Distributed file systems (DFSs) are another important category of file systems that need thorough checking. However, adopting local file system testing techniques for DFSs is difficult due to factors [124] like network communication, load redistribution, data replication, distributed concurrency, and scalability, which are not

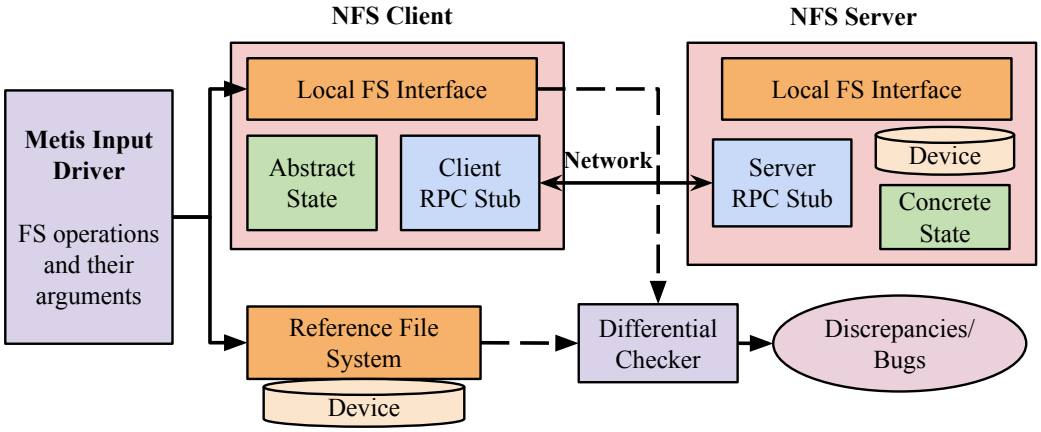


Figure 5.3: The structure of model checking NFS with Metis. We set up one client and one server on the same machine to simplify SPIN's management of concrete and abstract states. The local file system type used by NFS should be identical to the reference file system. Similar to Metis for other file systems, the differential checker compares the abstract state between the NFS client and the reference file system, and any discrepancy is considered a potential bug.

notable concerns in local file systems [56]. Although Metis is designed for local file systems, it has the potential to be applied to DFSs due to its general-purpose state definition and exploration method, as well as the flexibility of its differential checker [82]. As a classic example of a distributed file system, NFS (Network File System) [120] has been widely used for over 40 years and continues to be actively maintained and utilized today. Here, we present our efforts to use Metis to check two NFSv4 implementations: NFS kernel server [31] and NFS-Ganesha [103].

NFS has a client-server architecture where the server exports shared directories over the network, and the client mounts these remote directories, so that the client can access and perform file operations on them as if they were part of the local file system, with communication managed using RPCs (Remote Procedure Calls). NFS relies on a local file system (*e.g.*, Ext4) as backend storage on the server to store and manage files. Therefore, checking NFS primarily involves *examining the interaction between the server and clients*, as the local storage is handled by underlying local file systems like Ext4.

We extended Metis to check NFS using a simple setup, with one client and one server both running on the same machine and connected via the localhost network. Figure 5.3 illustrates the structure of model checking process for NFS in Metis.

Similar to how Metis checks local file systems, Metis generates test inputs (file system syscalls and their arguments) for both the NFS client (*i.e.*, the file system under test) and the reference file system. We selected the reference file system to be the same as the local file system used by NFS because it ensures that any differences detected in behavior are attributable solely to the NFS protocol rather than to inconsistencies between different (local) file system implementations.

The operations generated by Metis are executed on both the NFS client and the reference file system. Once the NFS client receives the syscall, it communicates with the NFS server, which processes the request on its local file system and returns the result to the client. This allows us to fetch the result value and error code from the client side and compare them against those from the reference file system. After each operation, we compute the abstract state on the NFS client side, which involves another round of network communication with the server. If there is a bug in the NFS protocol implementation, it can be detected through abstract state comparison in the differential checker, similar to how Metis checks local file systems. For state save/restore operations (SS/R) in NFS, the same challenge exists as with other local file systems—we cannot save the in-memory kernel state from a user process, so we must flush all memory to disk and checkpoint only the persistent disk state. Because all information is stored on the NFS server side, we access the concrete state by memory-mapping the NFS server's device and saving it with SPIN. It is worth noting that we attempted to use CRIU [26] to save and restore NFS-Ganesha's concrete state, given that NFS-Ganesha is a user-space server; however, this attempt was unsuccessful because it still depends on kernel-level resources.

5.5.2 NFS Checking Implementation and Discussion

Two file systems, Ext4 and RefFS, have been integrated as the reference and local NFS file systems for checking the NFS kernel server and NFS-Ganesha. We implemented different procedures for the two reference file systems when checking NFS due to differences in saving and restoring concrete states. While using Ext4, before each file system operation, we must first export the NFS server path, then mount it with Ext4, followed by mounting the client path. Conversely, after each operation, we have to unmount the client path, unexport the server path, and then unmount the server path to clear in-memory state and save the concrete state.

In contrast, RefFS can considerably simplify the process by virtue of its `ioctl` snapshot APIs. Using RefFS as the local file system eliminates the need for constant mounting and unmounting before and after each operation, as RefFS can

save and restore its entire state on its own. Without requiring mount/unmount operations, there is no need to export/unexport the NFS server path either, resulting in significantly better performance compared to Ext4 as the local file system. This highlights the advantages of RefFS in facilitating model checking not only for local file systems, but also its potential to enhance checking of distributed file systems.

We present here some preliminary evaluation findings related to the performance and bug detection of the NFS checking process. For performance evaluation, we used Metis to check kernel NFS with RefFS and Ext4 as the local file systems for NFS, respectively. During a 10-hour experiment, using RefFS as the local file system resulted in over 42 million file system operations and 11 million unique abstract states, with a processing rate of 1184.6 ops/sec and 306.5 states/sec. However, using Ext4 as the NFS backend yielded only 0.07 operations per second, significantly slower than using RefFS as the backend. This is because the constant mounting and unmounting of both the NFS client and server, as well as the repeated exporting and unexporting, take considerable time to complete even a single operation. Therefore, using Ext4 as the local file system for checking NFS in Metis is impractical, and we claim that RefFS is the suitable option for the task.

We observed two discrepancies in both NFS implementations, which turned out to be expected behaviors rather than real bugs. Consequently, we categorized them as false positives and handled them in our abstract state method. The first discrepancy we found is that for devices smaller than 1MB, NFS reports the size as 1MB instead of the actual device size. This difference is due to configuration settings. By adjusting the NFS `rsize` and `wsize`, we can obtain the correct size that reflects the actual backend device size. We identified a second discrepancy where temporary files were found on the NFS client but were absent from the reference file system. These temporary files are created by NFS when a file is deleted but still open by a process. We have modified our abstraction function to exclude them when computing the abstract state for NFS. Despite this, we believe that the Metis model checking approach with RefFS offers a promising method for checking distributed file systems like NFS. Further discussion and future work on model checking for distributed file systems can be found in Section 6.2.

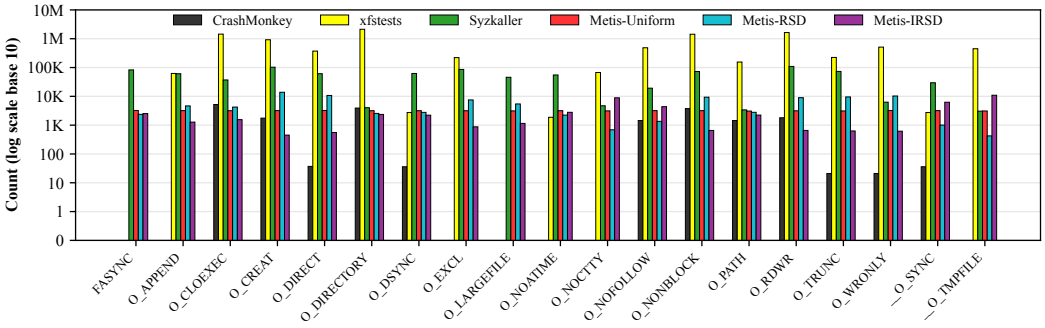


Figure 5.4: Input coverage counts (log_{10}, y-axis) of `open` flags (x-axis) for CrashMonkey, xfstests, Syzkaller, and Metis with 3 different weight distributions.

5.6 Evaluation

We evaluated the efficacy and performance of Metis and RefFS, specifically: **(1)** Does Metis have the versatility to test different input partitions compared to other testing tools? (See §5.6.1.) **(2)** What is Metis's performance? How does it scale with the number of VTs when using Swarm verification? (See §5.6.2.) **(3)** What is RefFS's performance compared to other file systems? How reliable and stable is RefFS, as Metis's reference file system? (See §5.6.3.) **(4)** With RefFS set as the reference file system, does Metis find bugs in existing Linux file systems? (See §5.6.4.)

Experimental setup We evaluated Metis on three identical machines, trying various configurations, particularly with multiple distributed VTs. Each machine runs Ubuntu 22.04 with dual Intel Xeon X5650 CPUs and 128GB RAM. We also allocated a 128GB NVMe SSD for swap space. We evaluated Metis's performance using RAM disks, HDDs, and SSDs by comparing Ext4 with Ext2. The results showed that RAM disks were 20× faster than HDD and 18× than SSD. Also, Metis performs best when the file system device is as small as possible. Therefore, we used RAM disks as backend devices for on-disk file systems and minimum mountable device sizes for all file systems in all evaluations that follow.

5.6.1 Test Input Coverage

We assessed input coverage (§5.2) for Metis and other file system tests on two dimensions: completeness and versatility. Completeness considers whether a testing

tool covers all input partitions (§5.2) in test cases. Versatility is the ability to tailor test cases for any desired input coverage. Metis outperforms existing checkers and a fuzzer [43] on both dimensions.

Comparison with existing testing tools We selected three testing tools, each representing a unique technique: CrashMonkey [98] for automatic test generation, xfstests [118] for (hand-written) regression testing, and Syzkaller [43] for fuzzing. To ensure fairness, we ran all of them and Metis (with one VT) to check Ext4 for 40 minutes each, because this time length was sufficient to complete all xfstests test cases and CrashMonkey's default test cases [99].

Measuring input coverage requires tracking the file system syscalls executed by the testing tool, including their associated arguments. Traditional syscall tracers (*e.g.*, `ptrace`-based ones) cannot distinguish the syscalls used on the file systems under test, because a testing tool makes many testing-unrelated syscalls, such as opening and reading dynamically linked libraries or logging statistics. CrashMonkey and xfstests do not inherently log their test inputs. Hence, we used a tool [83] specifically designed for measuring input coverage in file system testing to assess coverage for CrashMonkey and xfstests. Syzkaller's debug option and Metis's logger record all syscalls and arguments, enabling us to compute their input coverage using their internal mechanisms.

Input coverage for open flags Figure 5.4 shows the input coverage of `open`, partitioned by individual flags, for CrashMonkey, xfstests, Syzkaller, and Metis. In Metis, we set weights according to three input partition distributions: Uniform, RSD (Rank-Size Distribution [112]), and IRSD (Inverse Rank-Size Distribution [106]). Metis-Uniform denotes that Metis tests each input partition (*i.e.*, `open` flag) with a fixed weight (*i.e.*, probability). Both RSD and IRSD represent non-uniform distributions. We adopted the core principle of RSD, such that flags with higher ranks have higher test frequencies. Conversely, in IRSD, lower-ranked flags have higher frequencies. We analyzed the frequency of individual open flags' appearance in the 6.3 Linux kernel source. Metis employed those flags based on their proportional (Metis-RSD) and inverse-proportional (Metis-IRSD) frequencies. These distributions attempt to model two contrasting strategies: (1) Flags that appear more frequently in the kernel sources warrant proportionally more testing because they are used more frequently; conversely, (2) Flags with fewer occurrences in the kernel should be tested more thoroughly because they are more rarely used and hence could hide bugs for years.

In Figure 5.4, the x-axis labels every single-bit `open` flag and the y-axis (log_{10}) counts how often each was exercised by the testing tool. A higher y-value means more testing was conducted. We see that only Syzkaller and Metis covered all `open` flags. For instance, neither CrashMonkey nor xfstests tested the `O_LARGEFILE` flag, which could lead to missing related bugs [133]. Metis-Uniform test all flags equally; its coefficient of variation (CV) [1] (standard deviation as percentage of the mean) is only 1.2% (40-minute run). For its non-uniform test distributions, close examination of Figure 5.4 shows that `O_CREAT` (the most common `open` flag in the kernel source) is indeed tested most often in Metis-RSD and least in Metis-IRSD. `__O_TMPFILE`, the least-frequent flag, exhibits the opposite trend. Other tools lack the versatility to adapt their test input partitions to the desired amount of testing.

Moreover, we observed that xfstests tested certain input values (*e.g.*, `O_DIRECTORY`) millions of times while others (*e.g.*, `FASYNC`) are not tested at all. However, other tools sometimes have a higher total operation count than Metis because Metis has to unmount and remount the file system to achieve state tracking and verify state equality after each operation, slowing its syscall execution speed. Given the essential role of unmount/mount for state tracking (§5.3.2) and the need for state comparison (§5.3.3), we use Swarm verification to improve the overall operation efficiency (§5.3.5).

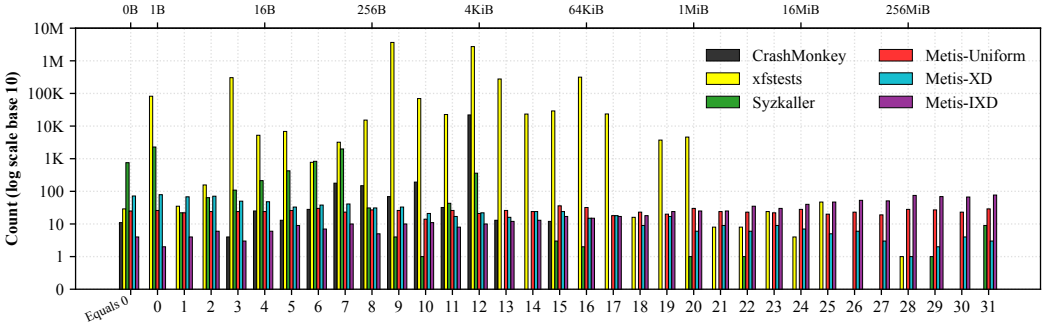


Figure 5.5: Input coverage (counts, log_{10}, y-axis) of `write` size (in bytes) for CrashMonkey, xfstests, Syzkaller, and Metis with three different weight distributions. The x-axis denotes the power of 2 of the write size (shown as $x2$-axis). Note a special "Equals 0" x-axis value for writes of size zero.

Input coverage for write size Figure 5.5 shows the input coverage for the `write` size (requested byte count). The x-axis represents the log_2 of the size,

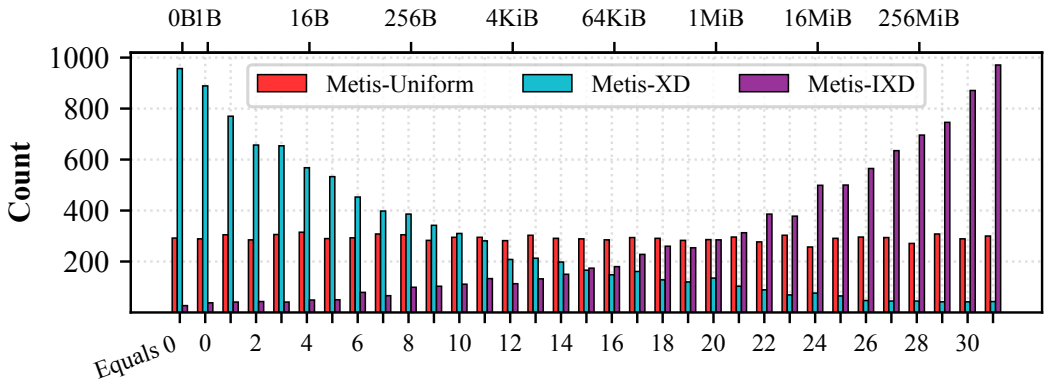
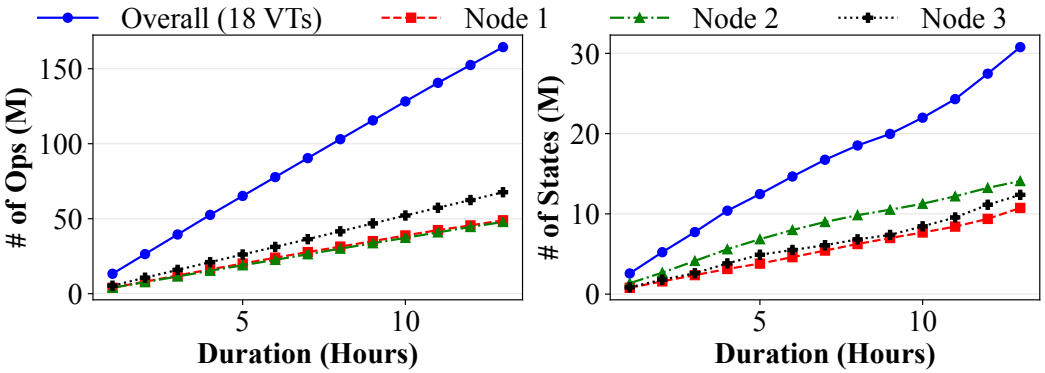


Figure 5.6: Input coverage of `write` size (in bytes) for Metis-Uniform, Metis-XD, and Metis-IXD, each running for 4 hours. The x-axis and $x2$-axis here are the same as in Figure 5.5, but the y-axis shows counts on a linear scale. As seen, with a longer run, the expected distributions are more accurate.

corresponding to the `write` size partitions (see §5.3.1). For example, $x = 10$ represents all sizes from 2^{10} to $2^{11} - 1$ (or 1024–2047). The y-axis (log_{10}) shows the number of times each x bucket was tested by a given tool. Only Metis ensured complete input coverage across all `write` size partitions. All other tools primarily tested sizes under 16MiB ($x \leq 24$). Certain partitions (*e.g.*, $x = 26$) were omitted by all these tools, even though systems with many GBs of RAM are now common. As with the `open` flags above, here Metis-Uniform also assigns uniform test probabilities to each write size partition. To illustrate Metis's versatility, we chose exponentially decaying distributions for write sizes. Metis-XD prioritizes testing smaller sizes more often, because they tend to be more popular in applications. The probability of each input partition is set to $0.9\times$ smaller than the previous one (in frequency order); all probabilities are then normalized to sum to 1.0. Metis-IXD emphasizes the inverse: testing input partitions with larger write sizes, on the hypothesis that they are less used by applications and thus latent bugs may exist. Here, the probability of each test partition is $0.9\times$ that of the next *larger* partition.

In Figure 5.5, the trend does not precisely align with the probabilities due to the relatively short 40-minute runtime and a correspondingly limited number of write operations, so the CV was 17.0%. When we ran Metis six times longer (4 hours), however, the CV dropped to 3.9% as seen in Figure 5.6; and when we ran it six times longer still (24 hours), the CV fell to a mere 2.6%. Due to space limitations, we omit showing the input coverage for other Metis-supported syscalls.



(a) File System Operations (b) Unique Abstract States

Figure 5.7: Metis performance with Swarm (distributed) verification, measured in terms of the number of operations and unique abstract states (in millions). Each node runs 6 VTs (one per CPU core), for a total of 18 unique VTs that collectively explored the state space. As seen, performance scales generally linearly with the number of VTs.

5.6.2 Metis Performance and Scalability

To evaluate performance with distributed Metis VTs, we deployed it on three physical nodes, comparing Ext4 (reference) to Ext2 (system under test) for 13 hours. Each node (machine) operated six individual VTs, totaling 18 VTs. Figure 5.7 shows the aggregate performance of the six VTs on each node, as well as the overall performance across all 18 VTs. We measured both file system operations (left) and unique abstract states (right). All VTs exhibited a linear increase in the number of operations executed over time. Over 13 hours, these 18 VTs executed more than 164 million operations, with each VT averaging 195 ops/s.

The count of explored states also increased steadily over time, although not exactly linearly. This is because executing operations does not always produce new, unseen states. For example, if a file exists, creating it again will not change the state. Thus, the number of unique states is fewer than the number of operations in a given time frame. Collectively, these VTs explored over 30 million unique states. On average, each explored 2.7 million states. Using 18 VTs resulted in exploring 11.2$\times$ more unique states than with a single VT. This experiment shows Metis's almost linear performance scalability with the number of VTs.

Different VTs might explore the same states, as each VT operates independently and without communicating with others. We evaluated the proportion of

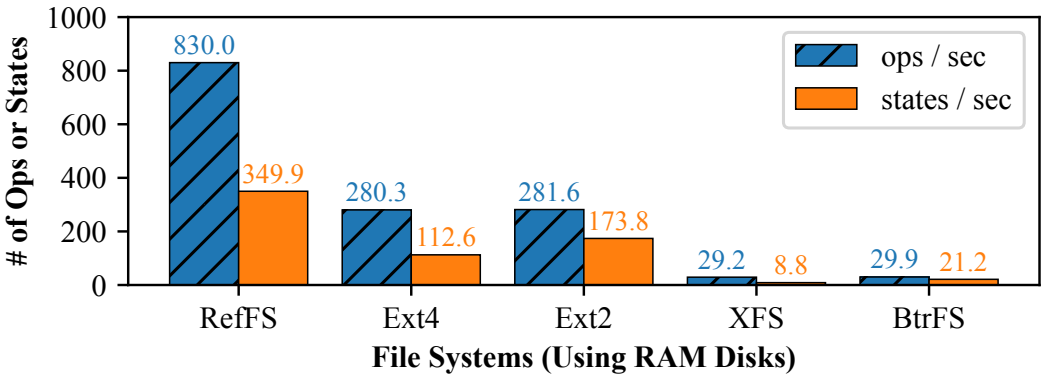


Figure 5.8: Performance comparison between RefFS and other mature file systems while being checked by Metis. The y-axis applies to both ops/sec and states/sec.

states explored by more than one VT, which represents "wasted" effort, a figure we want minimized. Our results showed that only about 1% of all states were duplicated across all VTs. Therefore, the redundancy of states explored by multiple VTs is relatively small and acceptable.

5.6.3 RefFS Performance and Reliability

To evaluate RefFS's performance, we used Metis to check it against a single file system. We also considered four other mature file systems (Ext4, Ext2, XFS, and BtrFS) as potential references. For a fair comparison, we use RAM disks as the backend devices and adopted the smallest allowed device size for each. Figure 5.8 shows that RefFS outperformed the others in terms of both operations and unique states per second. Even though RefFS is a FUSE file system—generally slower than in-kernel ones—it was 3.0×, 2.9×, 28.4×, and 27.7× faster than Ext4, Ext2, XFS, and BtrFS, respectively. This is primarily because Metis was able to use the save/restore APIs (§5.4.1) and thus did not have to unmount and remount RefFS.

Ext4 and Ext2 were faster than XFS and BtrFS due to the difference in minimum device sizes: the former require just 256KiB, whereas the latter need 16MiB. Mapping and copying larger devices in memory naturally increased time overheads.

Reliability To serve as a reference, RefFS must be highly reliable. While developing RefFS and Metis, we made necessary changes (110 lines of code) to xfstests

Bug#	FS	Causes & Consequences	D	C	N
1	BetrFS	Repeated mount and unmount caused a kernel panic	✔	✔	✔
2	BetrFS	`statfs` returned an incorrect `f_bfree`	✔	✔	✘
3	BetrFS	`truncate` failed to extend a file	✔	✔	✔
4	F2FS	A file showed the wrong size after another file was deleted	✘	✘	✔
5*	JFFS2	Data corruption occurred in a truncated file when writing a hole	✔	✔	✔
6	JFFS2	A deleted directory remained after unmounting	✘	✘	✔
7	JFFS2	GC task timeouts and deadlocks during operations	✔	✔	✘
8	JFS	NULL pointer dereference on `jfs_lazycommit`	✔	✘	✔
9	JFS	After writing to one file, another file's size changes	✘	✘	✔
10	NILFS2	NULL pointer dereference on `mdt_save_to_shadow_map`	✔	✘	✔
11	NILFS2	Failed to free space on a small device with cleaner	✔	✘	✔
12	NILFS2	Unmount operation hung after using `creat` on an existing file	✔	✘	✔
13	NOVA	Kernel hang due to improper snapshot cleaner kthread implementation	✔	✔	✔
14	NOVA	Incorrect file size after writing to a different file	✘	✘	✔
15	PMFS	Incorrect file size after creating a file	✘	✘	✔

Table 5.2: Kernel file system bugs discovered by Metis. In the table header, FS, D, C, and N represent the file system name, whether it is deterministic (D), confirmed (C), and new bug (N), respectively. This list excludes the 11 RefFS bugs that Metis detected and fixed. JFFS2 bug fix #5 (marked by *) was integrated into the Linux mainline recently.

so that we also could use it to debug RefFS. While we used xfstests to find certain bugs in RefFS, xfstests often misreported the bug information. For example, although we implemented RefFS's `link` operation, it still did not pass generic test #2, incorrectly indicating that the operation was unsupported. For that reason, we also used Metis to check RefFS with Ext4 as the reference. We discovered and fixed 11 RefFS bugs, aided by Metis's logs and replayer. Those bugs included failure to invalidate caches, inaccurate file size updates, erroneous `ENOENT` handling, and improper updates to `nlink`, among others. After fixing them, we evaluated RefFS against Ext4 using 18 distributed Metis VTs for 30 days, executing over 3.1 billion operations and exploring 219 million unique states. No discrepancies were reported, demonstrating that RefFS's reliability and robustness are similar to Ext4's—but with better performance when used as Metis's reference file system.

5.6.4 Bug Finding

With RefFS as our reference file system, we applied Metis to check seven existing file systems: BetrFS [62], BtrFS [115], F2FS [78], JFFS2 [134], JFS [60], NILFS2 [27], XFS [136], and two persistent memory file systems: NOVA [137], and PMFS [30], discovering potential bugs in seven. Table 5.2 summarizes these bugs, including causes and consequences, whether they were confirmed by developers, and whether they were new or previously known. Metis found bugs using both uniform and non-uniform input distributions, but some distributions found bugs faster. Some bugs were detected within minutes, while others took up to 22 hours, which is reasonable for long-standing bugs. The bugs we identified were not detected by xfstests [118] or Syzkaller [43]. Metis identified an F2FS bug that was not detected by Hydra [70]. We also checked file systems (*e.g.*, BetrFS) that are not currently supported by Hydra [70].

We found bugs using Metis through different indicators. Discrepancies reported by the differential checker accounted for nine out of fifteen detected bugs (# 2–6, 9, 11, 14, and 15). The remaining six caused a kernel panic (Linux "oops") or hung syscall (due to a deadlock). After analyzing each discrepancy using Metis's logger and replayer, we verified that all behavior mismatches originated from incorrect behavior in the file system under test—the reference file system, RefFS, was consistently correct.

We reported five bugs to BetrFS's and JFFS2's developers, all of which were confirmed as real bugs; however, one bug each in BetrFS and JFFS2 had already been fixed in the latest code base. Of the remaining unconfirmed bugs, four were deterministic and five were nondeterministic. Deterministic bugs are those easily

FS Testing Approach	**Input**	**FS effort**	**Ops effort**	**ST**	**CC**	**BD**
Metis: this work	👍👍👍	👎	👎	✔	✘	Behavioral discrepancies
Traditional Model Checking: CVFS [37], CREFS [143]	👍	👎👎👎	👎👎👎	✔	✘	User-specified assertions
Implementation-level Model Checking: FiSC [142], eXplode [141]	👍	👎👎	👎👎	✔	✘	User-written checkers
Fuzzing: Syzkaller [43], Hydra [70]	👍👍	👎👎	👎	✘	✔	External checkers
Regression Testing: xfstests [118], LTP [97]	👍	👎👎	👎👎👎	✘	✘	Preset expected outcome
Automatic Test Generation: CrashMonkey [98], Dogfood [20]	👍👍	👎	👎👎	✘	✘	External checkers or an oracle

Table 5.3: Comparison of representative file system testing tools. In the table header, Input, FS effort, Ops effort, ST, CC, and BD represent versatility to set test inputs, the effort required to test new file systems, the effort required to add new FS operations to testing, the ability to track states (state tracking, ST), the ability to track code coverage (CC), and the checker for bug detection (BD), respectively. In column 2, the more 👍 symbols, the more relatively versatile the system is; conversely, in columns 3–4, more 👎 symbols denote more effort.

reproducible after Metis reported a discrepancy or the kernel returned errors (*e.g.*, hang or `BUG`). We are currently pinpointing the faulty code for the deterministic bugs and preparing patches for submission to the Linux community. Metis also detected nondeterministic bugs that its replayer could not reproduce. For instance,

after using `unlink` to delete file `d-00/f-01`, the size of another file `f-02` in F2FS incorrectly changed to 0 instead of the correct value. Replaying the same syscall sequence did not reproduce this bug. To trigger it, we had to rerun Metis, but the time and number of operations needed varied across experiments. Given the bug's nondeterminism, we suspect a race condition between F2FS and other kernel contexts. We verified that these unconfirmed bugs persist in the Linux kernel repository (v6.3, May 2023) without any fixes, thus classifying them as unknown bugs.

To detect them, all these potential bugs require specific operations on a particular file system state, underscoring the value of both input and state exploration. JFFS2 bug #5 is an example of the interplay between input and state. After 4.3 hours of comparing JFFS2 with RefFS, Metis reported a discrepancy due to differing file content. We observed the bug occurred when truncating a file to a smaller size, writing bytes to it at an offset larger than its size, and then unmounting the file system to clear all caches. Uncovering this multi-step, data-corruption bug required specific inputs (`truncate`, `write`) and then unmounting and remounting, because there was a cache incoherency between the JFFS2 in-memory and on-disk states. Ironically, the fact that Metis was "forced" to un/mount, is exactly why we found this bug, which was present in the 2.6.24 Linux kernel and remained hidden for 16 years. We fixed this long-standing bug, and our patch has since been integrated into the Linux mainline (all stable and development branches).

5.7 Conclusion

File system development is difficult due to code complexity, vast underlying state spaces, and slow execution times due to high I/O latencies. Many tools and techniques exist for testing file systems, but they cannot be easily updated to test specific conditions at a configurable level of thoroughness. Moreover, they tend to require code or kernel changes or cannot easily adapt to testing new file systems.

In this paper, we presented Metis, a versatile model-checking framework that can thoroughly explore file-system inputs and states. Metis abstracts file-system states into a representation that can be used to compare the file system under test against a reference one. We designed and built RefFS, a reference POSIX file system with novel features that accelerate the model-checking process. When used with Metis, RefFS is 3–28$\times$ faster than other, more established, file systems. We extensively evaluated Metis's input and state coverage, scalability, and performance. Metis, helped by RefFS, can speed file-system development: we already

found a dozen bugs across several file systems. Overall, we believe that Metis, with its unique features, serves as a valuable addition to file system developers' tool suite. Finally, Metis's framework is versatile enough to be adapted to other systems (*e.g.*, databases).

Future work Our near-term plans include expanded state exploration using Swarm verification, investigating any bugs we discover, and then fixing and reporting them. We are also beginning to test network and distributed/parallel file systems [49].

In the long run, we plan the following: (i) Metis can trigger nondeterministic bugs, such as race conditions. Therefore, we need to integrate techniques to more deterministically explore and reproduce such bugs [41]. Also, we plan to explore kernel thread interleaving states to find more concurrency bugs [138]. (ii) We intend to enhance Metis by emulating crash states to identify crash-consistency bugs in kernel file systems [98, 76]. (iii) We aim to add support for testing controlled file-system corruptions [140, 49]. For example, if both RefFS and the test file system can be corrupted in a logically identical fashion, Metis can investigate more error paths (*e.g.*, those leading to `EIO`).

Chapter 6

Proposed and Future Work

In this chapter, we outline the proposed work (Section 6.1) that we intend to include in this thesis and then discuss potential future work (Section 6.2) that extends beyond the scope of this thesis.

6.1 Proposed Work

We introduce the proposed work in this section to describe the specific research tasks and methodologies we plan to incorporate into the thesis. This section explains two main proposed works: applying IOCov to improve the effectiveness of existing file system testing tools and implementing Containerized Swarm Verification (CoSV) to improve scalability and other aspects of Swarm model checking.

IOCov extension and application In this thesis proposal, we plan to extend our study of real-world file system bugs in several ways. First, in addition to using xfstests, we intend to use Syzkaller, a coverage-based OS fuzzer, to investigate the relationship between code coverage and test effectiveness. As Syzkaller is specifically optimized for code coverage and has successfully identified many bugs in Linux kernel file systems, this can gain a deeper understanding of effectiveness of the code coverage metric in file system testing. Second, we aim to pinpoint the specific inputs (syscalls and arguments) required to trigger and reproduce each bug and develop a clearer perspective on which inputs are most important for file system testing. This will help refine our definition and calculation of input/output coverage and thus result in more effective improvements to existing testing tools.

We propose to further develop and expand the IOCov framework based on the new insights gained from the bug study. We will enable IOCov to measure other inputs/outputs that are important for finding bugs but are not currently supported. Furthermore, in IOCov, we plan to support more syscalls, enhance our metrics to support bitwise combinations, explore non-uniform TCD target arrays, and support identifiers and pointer arguments. By doing so, IOCov can evaluate file-system testing tools under a broader range of scenarios, help balance test cases across input/output different partitions, prevent under-testing and over-testing, and more effectively capture key inputs and outputs that may be missed by existing testing tools.

Ultimately, the goal of IOCov is to enhance file system testing tools by optimizing input and output coverage to address under-testing and over-testing issues, thereby uncovering more hidden bugs in file systems. Therefore, we plan to enhance input/output coverage for specific file system testing tools (*e.g.*, CrashMonkey [98]) to enable more thorough testing of file systems and to detect more unknown bugs.

Scalable, containerized Swarm verification We have already demonstrated the benefits of using Swarm verification to parallelize Verification Tasks (VTs) over multiple CPU cores and multiple machines. However, applying Swarm verification faces several challenges, and the primary challenge is scalability. While we can deploy many VTs across multiple local machines, the original Swarm verification pre-generates VTs and scripts from a static configuration file and relies on SSH for VT transmission and execution, without proper orchestration. These constraints limit scalability in hybrid and dynamic environments (*e.g.*, hybrid cloud) because they lack mechanisms to monitor each VT's status, to allocate and deallocate VTs as conditions change, and to adapt to diverse hardware and software configurations (*e.g.*, different operating systems).

Two additional limitations exist concerning VT isolation and resource allocation. First, VTs running on the same machine lack isolation in their environments, potentially causing runtime interference and making it challenging to reproduce detected bugs deterministically. Second, the lack of fine-grained resource allocation and management may result in resource contention or underutilization among VTs on the same machine, ultimately leading to inefficiencies and waste. Therefore, we propose a Containerized Swarm Verification (CoSV) approach, wherein each Swarm verification VT operates in an isolated containerized environment, with orchestration used to facilitate the deployment and scaling of VTs.

CoSV offers multiple significant benefits over the vanilla Swarm verification:

1. **Scalability and Portability**: The primary benefit we aim to achieve is scalability: enabling Swarm verification to scale up more VTs within given computing and storage resources. With CoSV, each VT can be packaged independently and eliminate the need for SSH-based communication and file transferring. This way, with multiple machines and CPU cores, there is no need to manually set up SSH file transfer directories for VTs. CoSV ensures portability across heterogeneous environments to enhance scalability by encapsulating the entire software stack, making it easy to run verification task on any infrastructure, including different clouds, operating systems, or hardware. Additionally, it enables flexible configuration for various verification tasks, which can leverage modern containerization and orchestration techniques for ease of management.
2. **Environment Isolation**: Each VT operates in an isolated environment for model checking, which ensures consistency and allows for more deterministic reproduction of detected bugs. It also ensures security and fault isolation, preventing faults or downtime in one VT from affecting others.
3. **Fine-grained Resource Allocation**: CoSV provides finer-grained resource allocation for all VTs, allowing each VT to be allocated the precise computing resources (*e.g.*, CPU, RAM, Disk) it needs. This avoids resource contention and minimizes resource waste between VTs.
4. **Easy Deployment and Orchestration**: Many model checking applications (*e.g.*, Metis) have many prerequisites, dependencies, libraries, and configurations for each VT, which must be manually set up when using the original Swarm verification. Using containerization (*e.g.*, Docker [29]) and orchestration (*e.g.*, Kubernetes [73]) techniques, CoSV can encapsulate all prerequisites in a single effort and deploy them across different environments without compatibility issues, regardless of the underlying operating system or hardware.

We plan to compare the performance (*e.g.*, the number of unique states visited per second) and scalability of CoSV with Swarm verification, and ultimately apply CoSV to all Swarm verification tasks that could benefit from it. To demonstrate the scalability and ease of deployment of CoSV, we propose deploying CoSV applications in large-scale hybrid cloud environments such as Amazon Cloud [10] and Chameleon Cloud [68]. Furthermore, we will share the lessons we have learned that could be applicable to projects beyond SPIN.

6.2 Future Work

This section outlines future work by providing an overview of open research problems that can be derived from this thesis. These future works aim to enhance the capabilities of file system model checking from different perspectives, facilitate bug detection, and thus further improve file system reliability.

Model checking for distributed file systems (DFSs) The techniques in Metis can also be applied to distributed and network file systems. To demonstrate this, we have extended Metis to check both kernel NFS and NFS-Ganesha for NFS protocol versions 3 and 4 [119]. In our setup, the NFS client and server are configured on the same machine. We issue system calls and compute abstract states for the NFS client, ensuring that the client properly communicates with the server and allowing us to detect any bugs related to network connections.

However, this approach is insufficient for effectively finding bugs in distributed file systems (DFSs), as DFSs typically involve multiple nodes across different servers [111, 131, 94]. Currently, Metis and SPIN are not designed to compute abstract states (from clients) and concrete states (from servers) on separate machines. Therefore, network techniques need to be integrated to connect abstract and concrete states across different servers.

Moreover, fault injection techniques, such as network partitioning, node crashes, and device failures, are crucial for finding bugs in DFSs [49, 88], as some bugs only manifest during failure scenarios. Therefore, incorporating fault injection into the model checking of DFSs can be beneficial. Additionally, introducing faults into the backend local file systems of DFSs is worth exploring, with RefFS being a strong candidate for this role.

Model checking crash consistency and concurrency in file systems A promising direction for extending Metis is to enhance its ability to check file systems for more crash-consistency and concurrency bugs. For detecting crash-consistency bugs, in addition to the input and state exploration used in Metis, it is necessary to simulate and analyze the effects of crashes (*e.g.*, power failures, system shutdowns) on the file system's state and data integrity [23, 121]. On top of that, injecting simulated crashes at the appropriate points is essential to ensure that bugs can manifest (*e.g.*, after the `fsync` operation), and the crash state must be considered as part of the overall state description to explore as many unique crash states as possible [98, 76]. For detecting concurrency bugs, file system operations

should be executed concurrently using multiple threads. Additionally, checking file system concurrency requires incorporating thread interleaving states [41, 138] into the state representation and thoroughly exploring the states that could trigger concurrency bugs, such as race conditions, deadlocks, and other problems. The combination of concurrent syscalls and thread interleaving states significantly expands the state space, which requires more intelligent approaches to efficiently explore this vast space or prioritize the exploration of critical states.

Root cause analysis and reproduction for file system bugs Once a bug is found, it is equally important to reproduce it, identify the root cause, and fix it accordingly. However, based on our experience, there are certain types of bugs that can be detected due to their incorrect behavior or consequences but are difficult to reproduce. We refer to these as nondeterministic bugs. These nondeterministic bugs cannot be reliably reproduced using specific file system operations, states, or configurations, and we typically need to re-run the same syscalls multiple times to trigger the bug occasionally. Future work could focus on studying the characteristics of file system nondeterministic bugs and developing a method to consistently reproduce them. Along with bug reproduction, root-cause analysis plays a key role in resolving file system bugs [116]. Further exploration of root-cause analysis in conjunction with model checking is a worthwhile avenue of research.

Chapter 7

Conclusions

File systems play a crucial role in managing data storage and retrieval within operating systems. Given the potentially devastating effects of file system bugs, it is critical to develop techniques for bug detection. Unfortunately, existing testing techniques are inadequate to address the challenges posed by emerging file systems and hidden bugs as file system bugs continue to be reported regularly. We propose IOCov and Metis to tackle these challenges in two key ways: by developing new coverage metrics to evaluate and enhance existing testing tools, and by creating a new file system model checking framework for comprehensive file system validation.

We first identified that diverse syscall inputs and outputs are essential for triggering bugs during file system testing, based on a bug study. With many testing tools already available, we developed IOCov to compute input and output coverage for file system testing, as these metrics enable developers to enhance their tools more easily and effectively. We then implemented a file system model checking framework, Metis, to thoroughly explore the input and state space, using a differential checker that can detect a broader category of file system bugs with fewer restrictions. We also developed RefFS, a reference file system for Metis that is fast, lightweight, and reliable, optimized for state saving and restoration during model checking. Using Swarm verification, we deployed many parallel Metis verification tasks across multiple CPU cores and machines to collectively explore the state space and address the large state space problem.

Our preliminary results demonstrate that IOCov can detect missed test cases in existing file system testing tools and reveal both over- and under-testing problems. Metis is capable of generating thorough test inputs with flexibility and exploring file system states with near-linear scaling across multiple nodes. Additionally,

RefFS outperforms other mature file systems while acting as a reference in Metis. Metis aids in the development of RefFS by finding bugs and also detects many previously unknown bugs in other Linux kernel file systems.

Our thesis is that novel model checking techniques and coverage metrics are promising and can effectively help find file system bugs and improve file system reliability. We plan to apply IOCov to enhance existing testing tools by improving input coverage for detecting more file system bugs, and to use containers and orchestration systems to improve Swarm verification with better isolation, resource allocation, and easier deployment.

Bibliography

[1] Hervé Abdi. Coefficient of variation. *Encyclopedia of Research Design*, 1(5), 2010.

[2] Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. Code coverage analysis in practice for large systems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 736–745, Waikiki, Honolulu, HI, USA, May 2011. ACM.

[3] Alper Akcan. Fuse-ext2 GitHub repository, 2021. *https://github.com/alperakcan/fuse-ext2*.

[4] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. Re-animator: Versatile high-fidelity storage-system tracing and replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR '20)*, pages 61–74, Haifa, Israel, June 2020. ACM.

[5] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 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)*, pages 13–23, Porto de Galinhas, Brazil, April 2021. IEEE.

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

[7] Naohiro Aota and Kenji Kono. File systems are hard to test — learning from xfstests. *IEICE Transactions on Information and Systems*, 102(2):269–279, 2019.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.

[9] Algirdas Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[10] Amazon web services (aws). *https://aws.amazon.com/*.

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

[12] Ye Bin and Theodore Ts'o. Ext4: Fix potential out of bound read in ext4_fc_replay_scan(), 2022. *https://github.com/torvalds/linux/commit/1b45cc5c7b920fd8bf72e5a888ec7abeadf41e09*.

[13] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003. USENIX.

[14] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 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)*, pages 83–98, Atlanta, GA, April 2016. ACM.

[15] Lionel Briand and Dietmar Pfahl. 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)*, pages 148–157, Boca Raton, FL, USA, November 1999. IEEE Computer Society Press.

[16] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[17] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: An ultra-low latency and failure

resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[18] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association. Data set at *http://download.filesystems.org/auto-tune/ATC-2018-auto-tune-data.sql.gz*.

[19] Marsha Chechik, Benet Devereux, and Arie Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using SPIN. In *International SPIN Workshop on Model Checking of Software*, pages 16–36, Toronto, ON, Canada, May 2001. Springer.

[20] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. Testing file system implementations on layered models. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 1483–1495, Seoul, South Korea, June 2020. ACM.

[21] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017. ACM.

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

[23] Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, Eddie Kohler, and Nickolai Zeldovich. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

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

[25] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30, Elba Island, Italy, 2011. Springer.

[26] CRIU Community. Checkpoint/restore in userspace (CRIU), 2021. *https://criu.org/*.

[27] Benixon Arul Dhas, Erez Zadok, James Borden, and Jim Malina. Evaluation of Nilfs2 for shingled magnetic recording (SMR) disks. Technical Report FSL-14-03, Stony Brook University, September 2014.

[28] Felix Dobslaw, Robert Feldt, and Francisco de Oliveira Neto. Automated black-box boundary value detection. *arXiv preprint arXiv:2207.09065*, abs/2207.09065, 2022.

[29] Docker. *https://docker.com/*.

[30] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, pages 1–15, Amsterdam, The Netherlands, April 2014. ACM.

[31] Mike Eisler. NFS version 4. *https://www.usenix.org/legacy/event/lisa05/htg/eisler.pdf*.

[32] *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015. USENIX Association.

[33] Phyllis G. Frankl and Stewart N. Weiss. 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)*, pages 154–164, Victoria, British Columbia, Canada, October 1991. ACM.

[34] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.

[35] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores.

In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 100–115, Virtual Event / Koblenz, Germany, October 2021. ACM.

[36] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: adversarial memory and thread interleaving for detecting durable linearizability bugs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 195–211, Carlsbad, CA, July 2022. USENIX Association.

[37] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I. Siminiceanu. Model-checking the Linux virtual file system. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 74–88, Savannah, GA, USA, January 2009. Springer.

[38] Bernhard Garn and Dimitris E. Simos. 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*, pages 58–67, Cleveland, Ohio, USA, March 2014. IEEE Computer Society Press.

[39] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 302–313, Lugano, Switzerland, July 2013. ACM.

[40] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):1–33, 2015.

[41] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 66–83, Koblenz, Germany, October 2021. ACM.

[42] Google. KASan: Linux Kernel Sanitizers, fast bug-detectors for the Linux kernel, 2023. *https://github.com/google/kernel-sanitizers*.

[43] Google. Syzkaller: Linux syscall fuzzer, 2023. *https://github.com/google/syzkaller*.

[44] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 72–82, Hyderabad, India, May 2014. ACM.

[45] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 621–631, Minneapolis, MN, USA, May 2007. IEEE Computer Society Press.

[46] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 293–306, Stevenson, WA, October 2007.

[47] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 265–278, Cascais, Portugal, October 2011. ACM.

[48] Nitin Gupta. zram: Compressed RAM-based block devices, 2023. *https://www.kernel.org/doc/html/next/admin-guide/blockdev/zram.html*.

[49] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. A study of failure recovery and logging of high-performance parallel file systems. *ACM Transactions on Storage (TOS)*, 18(2):1–44, 2022.

[50] Nikolas Havrikov, Alexander Kampmann, and Andreas Zeller. From input coverage to code coverage: Systematically covering input structure with k-paths. Technical report, CISPA Helmholtz Center for Information Security, 2022.

[51] Hadi Hemmati. How effective are code coverage criteria? In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability*

and Security (QRS), pages 151–156, Vancouver, BC, Canada, August 2015. IEEE.

[52] Luis Henriques and Theodore Ts'o. Ext4: Fix error code return to user-space in ext4_get_branch(), 2022. *https://github.com/torvalds/linux/commit/26d75a16af285a70863ba6a81f85d81e7e65da50*.

[53] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[54] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[55] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2010.

[56] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[57] Atalay Mert Ileri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–338, Carlsbad, CA, October 2018. USENIX Association.

[58] Free Software Foundation Inc. Gcov, a test coverage program, 2023. *https://gcc.gnu.org/onlinedocs/gcc/Gcov.html*.

[59] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 435–445, Hyderabad, India, May 2014. ACM.

[60] JFS developers. Journaled file system technology for Linux, 2011. *https://jfs.sourceforge.net/*.

[61] Yujuan Jiang, Bram Adams, and Daniel M. Germán. 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)*, pages 101–110, San Francisco, CA, 2013. IEEE, IEEE Computer Society.

[62] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael A. Bender, Michael Condict, Alex Conway, Martín Farach-Colton, et al. BetrFS: A compleat file system for commodity SSDs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, pages 610–627, Rennes, France, April 2022. ACM.

[63] Dave Jones. Trinity: Linux system call fuzzer, 2023. *https://github.com/kernelslacker/trinity*.

[64] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In OSDI 2006 [104], pages 89–102.

[65] Natalia Juristo, Sira Vegas, Martín Solari, Silvia Abrahao, and Isabel Ramos. Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 330–339, Montreal, QC, Canada, April 2012. IEEE Computer Society Press.

[66] Wolfgang Kabsch. 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):922–923, 1976.

[67] Simon Kagstrom. KCOV: code coverage for fuzzing, 2023. *https://docs.kernel.org/dev-tools/kcov.html*.

[68] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the Chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 219–233. USENIX Association, Virtual Event, July 2020.

[69] Kernel.org Bugzilla. Ext4 bug entries, 2023. *https://bugzilla.kernel.org/buglist.cgi?component=ext4*.

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

[71] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, volume 1, pages 225–230, Ottawa, Canada, June 2007.

[72] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 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)*, pages 560–564, Montreal, QC, Canada, March 2015. IEEE Computer Society Press.

[73] Kubernetes. *https://kubernetes.io/*.

[74] Rick Kuhn, Raghu N. Kacker, Yu Lei, and Dimitris E. Simos. Input space coverage matters. *Computer*, 53(1):37–44, 2020.

[75] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[76] Hayley LeBlanc, Shankara Pailoor, Om Saran K. R. E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, pages 718–733, Rome, Italy, May 2023.

[77] Doug Ledford and Eric Sandeen. Bug 513221: Ext4 filesystem corruption and data loss, 2009. *https://bugzilla.redhat.com/show_bug.cgi?id=513221*.

[78] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In fast2015 [32], pages 273–286.

[79] Jerry Lee and Theodore Ts'o. Ext4: Continue to expand file system when the target size doesn't reach, 2022. *https://github.com/torvalds/linux/commit/df3cb754d13d2cd5490db9b8d536311f8413a92e*.

[80] David Leon and Andy Podgurski. 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)*, pages 442–453, Denver, CO, USA, November 2003. IEEE.

[81] The Linux Man-pages Project. *write(2) – Linux manual page*, 2023. *https://man7.org/linux/man-pages/man2/write.2.html*.

[82] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott Smolka, Wei Su, and Erez Zadok. 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)*, pages 123–140, Santa Clara, CA, February 2024. USENIX Association. Received all 3 Artifact-Evaluation badges.

[83] Yifei Liu, Gautam Ahuja, Geoff Kuenning, Scott Smolka, and Erez Zadok. 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)*, Boston, MA, July 2023. ACM.

[84] Yu Liu, Hong Jiang, Yangtao Wang, Ke Zhou, Yifei Liu, and Li Liu. Content sifting storage: Achieving fast read for large-scale image dataset analysis. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, July 2020. IEEE.

[85] Yu Liu, Yangtao Wang, Ke Zhou, Yujuan Yang, and Yifei Liu. Semantic-aware data quality assessment for image big data. *Future Generation Computer Systems*, 102:53–65, 2020.

[86] LTTng. LTTng: an open source tracing framework for Linux. *https://lttng.org*, April 2019.

[87] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '13)*, pages 31–44, San Jose, CA, February 2013. USENIX Association.

[88] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A fuzzing framework for distributed file systems. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)*, pages 529–543, Santa Clara, CA, July 2024.

[89] Filipe Manana. BTRFS: Fix NOWAIT buffered write returning –ENOSPC, 2022. *https://github.com/torvalds/linux/commit/a348c8d4f6cf23ef04b0edaccdfe9d94c2d335db*.

[90] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 218–233, Shanghai, China, October 2017. ACM.

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

[92] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[93] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. CNSBench: A cloud native storage benchmark native storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual, February 2021. USENIX Association.

[94] Sun Microsystems. Lustre file system: High-performance storage architecture and scalable cluster file system white paper. *www.sun.com/servers/hpc/docs/lustrefilesystem_wp.pdun.com/servers/hpc/docs/lustrefilesystem_wp.pdf*, December 2007.

[95] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 361–377, Monterey, CA, October 2015. ACM.

[96] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 499–510, Vancouver, BC, Canada, June 2008. ACM.

[97] Subrata Modak. Linux test project (LTP), 2009. *http://ltp.sourceforge.net/*.

[98] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

[99] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey: tools for testing file-system reliability, 2023. *https://github.com/utsaslab/crashmonkey*.

[100] Madanlal Musuvathi, Andy Chou, David L. Dill, and Dawson R. Engler. Model checking system software with CMC. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 219–222, Saint-Emilion, France, July 2002. ACM.

[101] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, December 2002. USENIX Association.

[102] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 57–68, Chicago, IL, USA, July 2009. ACM.

[103] NFS-Ganesha, 2016. *http://nfs-ganesha.github.io/*.

[104] *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006. ACM SIGOPS.

[105] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[106] Can Özbey, Talha Çolakoğlu, M Şafak Bilici, and Ekin Can Erkuş. A unified formulation for the frequency distribution of word frequencies using the inverse Zipf's law. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 1776–1780, Taipei, Taiwan, July 2023. ACM.

[107] Brandon Philips. The fsck problem. In *The 2007 Linux Storage and File Systems Workshop*, 2007. *https://lwn.net/Articles/226351/*.

[108] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 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)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

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

[110] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. System call clustering: A profile-directed optimization technique. Technical report, The University of Arizona, 2002.

[111] Glusterfs. *http://www.gluster.org/*.

[112] William J. Reed. On the rank-size distribution for human settlements. *Journal of Regional Science*, 42(1):1–17, 2002.

[113] Stuart C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings of the Fourth International Software Metrics Symposium (METRICS)*, pages 64–73, Albuquerque, NM, USA, November 1997. IEEE Computer Society Press.

[114] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 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)*, pages 38–53, Monterey, CA, October 2015. ACM.

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

[116] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009. ACM.

[117] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pages 167–182, Vancouver, BC, Canada, August 2017. USENIX Association.

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

[119] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. RFC 3530, Network Working Group, April 2003.

[120] S. Shepler, M. Eisler, and D. Noveck. NFS version 4 minor version 2 protocol. RFC 7862, Network Working Group, November 2016.

[121] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[122] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. 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, pages 409–424, Tallinn, Estonia, March 2012. Springer.

[123] Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. Model-checking support for file system development. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*, pages 103–110, Virtual, July 2021. ACM.

[124] Alexander Thomson and Daniel J Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In fast2015 [32].

[125] Yuan Tian, Julia Lawall, and David Lo. Identifying Linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 386–396, Zurich, Switzerland, June 2012. IEEE Computer Society Press.

[126] Linus Torvalds. Linux kernel source tree, 2023. *https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git*.

[127] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 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)*, pages 1–16, London, United Kingdom, April 2016. ACM.

[128] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Semi-valid input coverage for fuzz testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 56–66, Lugano, Switzerland, July 2013. ACM.

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

[130] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, pages 127–143, Rome, Italy, May 2023. ACM.

[131] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In OSDI 2006 [104], pages 307–320.

[132] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703, 1991.

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

[134] David Woodhouse, Joern Engel, Jarkko Lavinen, and Artem Bityutskiy. JFFS2, 2009.

[135] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. DEVFUZZ: automatic device model-guided device driver fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP)*, pages 3246–3261, San Francisco, CA, May 2023. IEEE.

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

[137] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 478–496, Shanghai, China, October 2017. ACM.

[138] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, pages 1643–1660, Virtual Event, November 2020. IEEE.

[139] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313–2328, Dallas, TX, October 2017. ACM.

[140] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In

Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland), pages 818–834, San Francisco, CA, May 2019. IEEE.

[141] Junfeng Yang, Can Sar, and Dawson Engler. 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)*, pages 131–146, Seattle, WA, November 2006. USENIX Association.

[142] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, December 2004. ACM SIGOPS.

[143] Jingcheng Yuan, Toshiaki Aoki, and Xiaoyun Guo. Comprehensive evaluation of file systems robustness with SPIN model checking. *Software Testing, Verification and Reliability*, 32(6):e1828, 2022.

[144] Insu Yun. *Concolic Execution Tailored for Hybrid Fuzzing*. PhD thesis, Georgia Institute of Technology, December 2020.

[145] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(2):161–196, 2006.

[146] Andreas Zeller, Holger Cleve, and Stephan Neuhaus. Delta debugging: From automated testing to automated debugging, 2023. *https://www.st.cs.uni-saarland.de/dd/*.

[147] Duo Zhang, Om Rameshwar Gatla, Wei Xu, and Mai Zheng. A study of persistent memory bugs in the Linux kernel. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR)*, pages 1–6, Haifa, Israel, June 2021. ACM.

[148] Zhiqiang Zhang, Tianyong Wu, and Jian Zhang. Boundary value analysis in automatic white-box test generation. In *Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 239–249, Gaithersbury, MD, USA, November 2015. IEEE Computer Society Press.

[149] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 259–274, Huntsville, ON, Canada, October 2019. ACM.