Advancing File System Model Checking: Coverage, Framework, and Scalability

A Dissertation 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

August 2025

Technical Report FSL-25-02

Stony Brook University
The Graduate School

Yifei Liu

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

Erez Zadok - Dissertation Advisor
Professor, Department of Computer Science

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

Dongyoon Lee
Associate Professor, Department of Computer Science

Scott A. Smolka
Professor, Department of Computer Science

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

This dissertation is accepted by the Graduate School

Celia Marshik
Dean of the Graduate School

Abstract of the Dissertation

Advancing File System Model Checking: Coverage, Framework, and Scalability

by

Yifei Liu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2025

File systems serve as the foundation for data storage and access, making their reliability crucial to maintaining system correctness and data integrity. However, building robust file systems remains a significant challenge. Despite numerous testing and verification techniques, file system bugs continue to emerge. To detect file system bugs and improve reliability, we tackle three key aspects: new coverage metrics for testing, a novel model checking approach, and enhanced scalability for file system verification. We begin by introducing input and output coverage (IOCov) as metrics for evaluating and improving file system testing, along with IOCov to compute them. We integrated IOCov into existing file system testing workflows, achieving broader input coverage and improving the detection of crash consistency bugs. Next, we present Metis, a file system model checking framework designed to explore diverse inputs under different file system states. Using a reference file system (RefFS), Metis compares the behaviors of two file systems and reports any discrepancies as potential bugs. Metis leverages Swarm Verification (SV) to scale state exploration by distributing parallel verification tasks (VTs) across multiple cores and machines. Finally, we describe Containerized Swarm Verification (CoSV), in which each VT runs in a container and is managed by an orchestrator. CoSV enhances the scalability of SV by packaging each VT as a self-contained unit, allowing for easy adaptation to dynamic resource availability.

In addition, CoSV ensures fault isolation across VTs to prevent faults in one task from interfering with the execution of others.

Our thesis is that effective file system testing requires coverage metrics to guide evaluation, new techniques for thorough checking, and scalable parallelism to explore large state spaces. Overall, input/output coverage helps developers evaluate file system testing, while model checking systematically verifies states, and containerized swarm verification scales this process through efficient, fault-isolated parallelism.

To my family, for being my greatest source of strength.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

Pursuing a Ph.D. demands substantial and enduring dedication and perseverance, marked by years of intensive work, persistent challenges, and inherent uncertainty. Nonetheless, with the support and encouragement I received, this journey has been deeply rewarding and fulfilling. I would not have reached this point, nor completed this thesis, without the many individuals who stood by me along the way.

First of all, I would like to express my sincerest gratitude to my advisor, Prof. Erez Zadok, for his unwavering guidance throughout the course of my Ph.D. program. Prior to beginning my Ph.D., I was already engaged in storage research and familiar with Prof. Zadok's influential work in the field. His research inspired me to apply to Stony Brook University with a clear goal of conducting research in computer storage systems. Becoming a member of the File systems and Storage Lab (FSL) was one of the most exciting and meaningful moments of my doctoral studies. Prof. Zadok has guided me through every aspect of research and shown me what it takes to become both a better researcher and a better person. He has mentored me through every stage of my Ph.D., from technical tasks such as coding and benchmarking to essential skills like academic writing, communication, presentation, and leadership. I am immensely grateful to have had an advisor who cared for me at all times, corrected me when I made mistakes, showed patience when I was slow, and did everything he could to support me when I struggled.

I thank Professors Geoff Kuenning and Scott Smolka for their mentorship, collaboration, and steadfast support over the years. I learned a great deal about computer systems, critical thinking, and English language skills from Prof. Kuenning, who was always available to help, offer insightful ideas, and suggest the optimal solutions among various possible directions. Prof. Smolka taught me the fundamentals of model checking and formal verification, and helped me learn how to analyze problems systematically and identify logical flaws in my reasoning. His willingness to challenge my assumptions played a key role in shaping my analytical mindset. I would also like to thank Dr. Gerard Holzmann, a renowned computer

scientist and a leading expert in software verification. Whenever I have questions regarding Spin model checking or Swarm verification, Dr. Holzmann is the person I turn to for advice. He has been a role model and a source of inspiration for my work on file system model checking and my pursuit of high-impact research.

I thank Professors Omar Chowdhury and Dongyoon Lee for serving on my dissertation committee and for providing insightful comments and constructive feedback on my work. Interacting with Prof. Chowdhury has always been intellectually enriching, as I not only learned about state-of-the-art developments in various areas but also acquired perspectives that helped improve my own work. Taking Prof. Lee's operating systems course in my first semester laid the foundation for my understanding of the Linux kernel, which has been highly useful for my research projects throughout the years.

I am grateful to both my FSL lab-mates and external collaborators for their support and contributions. Wei Su and I worked together on many aspects of the file system model checking project. Tyler Estro guided me on the multi-tier caching project when I first joined the lab. I also thank the other collaborators: Prof. Mário Antunes, Prof. Anshul Gandhi, Dr. Carl Waldspurger, and Prof. Avani Wildani, for their advice and input during the early years of my Ph.D. work on this project. I would like to acknowledge the students from FSL with whom I have had the pleasure of working, including: Manish Adkar, Gautam Ahuja, Rohan Bansal, Kevin Cai, Gomathi Ganesan, Tejeshwar Gurram, Ajay Hegde, Jerry Kang, Pei Liu, Shushanth Madhubalan, Gautham Narendra, Md. Kamal Parvez, Tanya Raj, Divyaank Tiwari, and Haolin Yu. I am also thankful for the opportunity to work alongside the following undergraduate students from Harvey Mudd College: Alex Bishka, Erin Li, Dan Pasiada, and Yongyi Wang. Even though we did not work together directly, I greatly benefited from discussions and support from the following FSL labmates: Ibrahim "Umit" Akgun, Alex Merenstein, Christopher Smith, and Maliha Tabassum.

In addition, I extend my appreciation to my friends, among them Ph.D. peers and others, who have supported me both technically and emotionally. During the pandemic and other difficult times, their presence made me feel less alone and helped me face every obstacle with courage. Here are just a few of them: Lu Chen, Gaofeng Deng, Heng Fan, Xi Han, Runzhou Han, Ping Huang, Le Hou, Ruyi Lian, Wenjie Liu, Rui Liu, Tong Liu, Qiaomu Miao, Yan Ma, Jiaxiang Ren, Tao Sun, Junyi Tao, Shilei Tian, Cong Wang, Jingyi Xu, Wei Xu, Yilun Wu, Wentao Wu, Yunting Yin, Caitao Zhan, Xinyu Zhang, and Puyang Zheng.

I owe a profound debt of gratitude to my family. My parents have supported me every step of the way since I was born. Though they may not have fully understood

my work, they never hesitated to trust their son, to respect my choices, and to lift me up when I was feeling down. My parents-in-law have consistently believed in me and supported me from afar, and I am truly appreciative of their constant faith. I also thank my sister, Lan, who guided me in choosing Computer Science as my major when I was uncertain, ultimately leading me to a field to which I am strongly committed and about which I am genuinely passionate. I would like to thank my cousin, Weihua, whose invaluable advice on academics and life helped steer me in the right direction.

Finally, and most importantly, words cannot fully express how grateful I am to my wife, Qinyun, for her love, compassion, and wholehearted belief in me. Among all the unforeseen gifts of graduate school, none has meant more than meeting Qinyun. Falling in love with her has imbued this chapter of my life with warmth and meaning. Qinyun is a wonderful life partner, a brilliant data scientist, and the most understanding person I know. I will always cherish the time we spent together, the countless delicious meals she prepared with thoughtful care, and the places we wandered hand in hand. With her by my side, the future feels filled with hope and touched by the whisper of blessings yet to unfold.

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 [28], cloud storage [130, 52, 53, 7], big data processing [131, 214, 132, 192, 122, 175], and virtualization technologies [140]. 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 [168, 182, 127, 124]. Although many file system testing techniques have been developed, new bugs continue to surface [107]. Compounding the issue is the emergence of new file systems that often receive limited testing, posing challenges to traditional techniques [200, 50, 123, 124].

There are three directions to address these challenges: (1) improving existing testing through better coverage, (2) developing new checking techniques for easier and more comprehensive file system validation, and (3) enhancing test scalability through increased parallel execution on computing resources. All three aim 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 [109]. Given the distinct semantics of file systems such as persistent state, POSIX compliance, and crash consistency, there is a need to develop new and effective coverage metrics [92] specifically designed for file system testing.

Second, new techniques for testing file systems should be developed to enable more comprehensive test cases and better coverage for uncovering bugs, while minimizing constraints such as the need for kernel instrumentation or modification when testing newly emerging file systems [200, 107].

Third, the complexity of file systems and the need to validate many properties and states make thorough testing computationally demanding. Scalable testing enabled by parallel execution is essential to efficiently explore large file system state spaces, reduce testing time, and uncover subtle bugs that might otherwise go undetected [123, 124, 127, 182].

Nevertheless, each of the three aspects—coverage metrics, testing methodology, and scalability—requires 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 underinvestigated [128, 129]. Additionally, even though the developer knows which lines were not covered, it is challenging to modify tests to cover them [8, 2].

Different tools have been developed for testing file systems using various approaches, but new bugs continue to emerge on a regular basis [107, 203, 106]. The limitations of these tools stem from three main factors: poor coverage [8], difficult applicability to new file systems [203], and the effectiveness of bug-finding checkers [206]. In terms of coverage, some testing tools are manually developed [172, 150] but they struggle to exercise enough cases across file systems; other tools fail to properly handle corner cases involving diverse operations and uncommon file system states [207, 206, 127, 182, 123, 124]. Regarding applicability, as new file systems emerge, testing tools frequently face challenges adapting to them because of inherent design differences [115, 200]. As for the effectiveness of checkers, different bug types require different tools (*e.g.*, POSIX [168] and crash consistency checkers [151]), and using ineffective ones can leave bugs undetected.

Scalability remains a core limitation in many existing testing tools. Many rely on single-threaded or sequential execution [206, 107]; this significantly limits their time efficiency and ability to check sufficient file system behaviors, states, and corner cases within a practical time-frame.

In this thesis, we address challenges in file system testing by leveraging coverage metrics and enabling more thorough, versatile, and scalable checking through three main thrusts: (1) designing new coverage metrics that assess and 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 file systems; and (3) improving the scalability of file system model checking through containerization and orchestration of swarm verification.

We achieve the first thrust by conducting a study of existing file system bugs in Ext4 and Btrfs to evaluate the effectiveness of the most widely used coverage metric in file system testing: code coverage [2]. Our findings indicate a weak correlation between code coverage and test effectiveness in file system testing. Based on our

bug analysis, we formulate two new coverage metrics, input coverage and output coverage, to evaluate file system testing tools by examining the coverage of system call inputs and outputs. We also developed IOCov, a framework for computing input and output coverage of file system testing tools. Using IOCov, we enhanced the input coverage of the crash consistency testing tool CrashMonkey [151], resulting in CM-IOCov, which retains the original design while achieving higher input coverage. We show that IOCov uncovers untested inputs, outputs, and value ranges in existing tools, and that CM-IOCov improves bug detection through enhanced input coverage. Notably, CM-IOCov detected bugs that the original CrashMonkey failed to find on both Linux 6.12 and 5.6.

To achieve the second thrust, we created Metis, a novel model-checking framework that enables thorough and versatile input and state space exploration of file systems. Metis combines the exhaustive state-space exploration of model checking with the cross-implementation validation of differential testing to check file systems without requiring an abstract model. This design allows users to check file systems without deep expertise in file systems or model checking and eliminates the need to create a model for each new file system. We also developed RefFS, a fast, lightweight, and reliable reference file system designed to accelerate model checking and enhance bug reproducibility through innovative `ioctl` APIs. Using RefFS as the reference, Metis successfully identified over 15 bugs across multiple file systems.

To realize the third thrust, we integrated Swarm verification [86] into Metis to enable parallel exploration across multiple CPU cores and machines. Moreover, we designed CoSV to use containers for packaging verification tasks (VTs), which are managed by an orchestrator [147]. We implemented two CoSV variants using different container technologies: Docker [47] and Kata Containers [104], referred to as CoSV-Docker and CoSV-Kata, respectively. CoSV streamlines packaging with a single effort and enables easy deployment of VTs as computing resources are added for improved scalability. CoSV uses different containers to provide varying fault isolation: CoSV-Docker handles resource contention among VTs, while CoSV-Kata isolates kernel-level faults caused by Metis or other low-level model checking.

It is our thesis that input and output coverage metrics are essential for effective file system testing, and that new model checking techniques should be employed to enable more thorough, scalable, and more easily applicable validation across diverse file systems. To realize our thesis, we first introduce new coverage metrics to evaluate and improve file system testing, along with IOCov for efficiently computing these metrics and CM-IOCov that demonstrates enhanced input coverage

for crash consistency testing. We then develop the model-checking framework Metis and the reference file system RefFS to enable more thorough and effective testing of file systems. Finally, we present CoSV, which enhances Swarm verification to improve the scalability of model checking task deployment, enabling the exploration of large state spaces such as those in file systems.

To promote reproducibility and future research, we have open-sourced most of the artifacts associated with this dissertation, including:

- IOCov [128, 129]: *https://github.com/sbu-fsl/IOCov*
- CM-IOCov [128, 129]: *https://github.com/sbu-fsl/CM-IOCov*
- Metis [125, 127]: *https://github.com/sbu-fsl/Metis*
- RefFS [126, 127]: *https://github.com/sbu-fsl/RefFS*

The rest of the thesis 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, the IOCov framework, and CM-IOCov as a practical application of input coverage. Chapter 5 presents the Metis model checking framework and the RefFS reference file system. Chapter 4 elaborates on CoSV and how it enhances Swarm verification, and Chapter 7 concludes this thesis and discusses directions for future research.

Chapter 2

Motivation

In this chapter, we describe the vision and motivations behind this thesis. Specifically, we aim to improve the effectiveness and scalability of file system model checking by: developing new coverage metrics to better evaluate and guide file system testing, designing a new model checking framework tailored for file systems to enable systematic and efficient checking, and enhancing the scalability of Swarm verification to enable broader exploration of complex system behaviors.

2.1 Thesis Statement

The problems addressed in this thesis are threefold: (1) to define and compute effective coverage metrics for file system testing that facilitate bug detection; (2) to develop a new file system model checking framework that overcomes limitations of existing tools, such as inadequate coverage and poor adaptability to emerging file systems; and (3) to enable scalable swarm verification for file system model checking and other tasks by concurrently exploring large state spaces and improving verification efficiency.

To address the first problem, this thesis introduces input and output coverage metrics for file system testing, along with the IOCov framework for computing these metrics and CM-IOCov for applying them to improve crash consistency testing in practice. To tackle the second problem, this thesis presents the Metis model checking framework, which incorporates the RefFS reference file system to integrate implementation-level model checking with the concept of differential testing for detecting bugs in Linux kernel file systems. To resolve the third problem, this thesis presents CoSV, an improved version of Swarm verification that leverages

containers and an orchestrator to improve scalability.

The goal of this thesis is to streamline the entire process of file system model checking—ranging from coverage and framework design to scalability—to assist developers in identifying hard-to-detect bugs 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 [102]. 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 [172]) typically focus on functionality coverage, seeking to test as many functions and features of the file system as possible. Some black-box testing tools [151, 31], 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 [68, 201, 107]. 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, we explore the effectiveness and usability of code coverage and introduce input and output coverage metrics that offer both high effectiveness and usability in file system testing. We also demonstrate that improving input coverage alone can help uncover more bugs and enhance test effectiveness.

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 [39]. 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 [207, 206]. 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 [40]. Given this, model checking has been successfully applied to identify many 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, we aim to address the limitations of existing file system model checking and fully explore the potential of model checking techniques [123, 124, 182, 127]. 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 [72] 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.

2.4 Scalability and Fault Isolation in Swarm Verification

Swarm Verification (SV) [84, 86] is a technique for the SPIN model checker [79] that generates multiple parallel verification tasks (VTs) by varying search parameters, causing each VT to explore different portions of the state space. SV aims to utilize available computing resources, including CPU cores and machines, to enable collective and parallel state-space exploration. Therefore, the scalability of VTs directly affects the efficiency and practicality of SV across various model checking tasks. However, SV focuses on generating multiple VTs but lacks efficient mechanisms to deploy them at scale on computing resources, particularly

when resource availability is dynamic. When model checking tasks involve complex configurations and dependencies, SV requires environment setup on each deployment machine, limiting scalability due to high manual effort. Moreover, SV executes VTs on the same machine without isolation, making it possible for faults in one VT to affect others, which is a frequent concern in model checking scenarios.

In this thesis, we address scalability and fault isolation in SV through containerization, packaging each VT in a container managed by an orchestrator. Different containers offer varying levels of isolation and performance, making them suitable for different types of model checking. Standard containers like Docker [47] are lightweight and offer better performance, but provide limited isolation. This is suitable for most user-space model checking tasks, which do not require isolation at the operating system kernel level. However, for model checking low-level software such as file systems, which may trigger kernel-level errors, sandboxed containers like Kata Containers [104] provide stronger isolation by running each container with a separate operating system kernel, at the cost of reduced performance. We implemented both variants of containerized swarm verification to evaluate their scalability and fault isolation compared to standard SV in realistic model checking scenarios.

Chapter 3

Related Work

In this chapter, we survey related work on test coverage metrics, file system testing and debugging, verified file systems, and distributed parallel model checking. Section 3.1 reviews related work on test coverage metrics in file system testing and prior studies evaluating the effectiveness of code coverage. Section 3.2 surveys related work on file system testing for bug detection, as well as verified file systems that employ formal techniques to ensure correctness and reliability. Section 3.3 discusses related work on parallel techniques for SPIN and swarm verification, as well as parallel and distributed approaches to model checking.

3.1 Test Coverage Metrics

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

Test Coverage Metrics Coverage metrics have long been a cornerstone of software testing, providing a quantitative method to evaluate the thoroughness of test suites [217, 138]. There are general coverage metrics for most software and specialized ones for specific test targets [164]. As the most widely used general metric, code coverage includes subtypes such as line, statement, function, and branch coverage, categorized by the granularity of the code measured [94]. In file system testing, however, developers primarily conduct tests by issuing syscalls to file systems in kernel space. Due to the complex path between user-space test suites and kernel-space file system implementations, it is unclear which syscalls to issue to cover specific code [60, 8].

Specialized coverage metrics [204] differ across testing techniques because each technique targets different bug types, exercises different system features, and uses distinct methods to generate and run test cases. For example, file system model checking [127, 206] aims to achieve state coverage by exploring as many file system states as possible. Similarly, testing crash consistency requires one to cover diverse crash scenarios [151, 23, 161, 5]. Current approaches to file system testing primarily rely on generic coverage metrics, such as code coverage, without specifically evaluating their effectiveness for this domain [107]. Moreover, no formal coverage metrics have been defined explicitly for file system testing.

Although input and output coverage metrics exist [112, 6, 189], they are designed for different targets (*e.g.*, quantum programs, network protocols) and are not suited for file system testing, which relies on syscall inputs and outputs.

Effectiveness of Code Coverage Assessing the effectiveness of code coverage is an active area, but the findings remain inconclusive and lack consensus. We define *effectiveness* as the ability to detect faults or bugs in the test target. Some studies [109, 64, 65] suggest that the effectiveness of code coverage is inconsistent and depends on the specific target. For example, Kochhar *et al.* [109] found a strong correlation between code coverage and test effectiveness for the Mozilla Rhino JavaScript engine, whereas the correlation was moderate for Apache HTTPClient. Some studies [56, 57, 69, 77] show that the correlation is contingent upon specific subclass metrics of code coverage. According to Gopinath *et al.* [69], statement coverage is the most effective metric for detecting faults, outperforming other code coverage metrics. Hemmati *et al.* [77] observed, however, that statement coverage is significantly weaker than other coverage metrics such as branch coverage.

Moreover, prior work [155, 25, 27, 92] indicates that code coverage has a limited correlation with test effectiveness, and thus new, complementary coverage criteria are needed [179]. Specifically, Inozemtseva *et al.* [92] analyzed 31,000 test suites for five systems and found a low-to-moderate correlation between code coverage and effectiveness. Nevertheless, there is no existing research examining this correlation for complex low-level systems, such as in-kernel file systems. This work investigates the correlation for file system testing through the bug study presented in Section 4.2.

File System Testing File system testing can be static or dynamic, depending on whether it involves actively exercising the file system [149]. This thesis focuses on dynamic testing, which generally has three steps: (1) generating test cases in

the form of syscalls, (2) initializing the file system under test and executing the tests, and (3) validating file system properties post-execution. Here, we consider four representative methods: regression testing, model checking, fuzzing, and automatic test generation, and explain how they achieve coverage.

Regression testing (*e.g.*, xfstests [172] and LTP [150]) is a collection of tests manually crafted to ensure that updates or new features do not introduce bugs or failures. It often aims to achieve functionality coverage by testing as many file system features as possible to verify the correctness of each. Given, however, the continuous evolution of file system features and the handcrafted nature of regression testing, it is difficult to ensure complete functionality coverage or provide a measure to assess the completeness of the testing [8].

Model checking [127, 182, 207, 206] is a formal verification technique used to find bugs by automatically and systematically exploring a file system's state space. During exploration, model checking verifies whether each file system state adheres to a specification. State coverage, however, is not a practical metric for file system testing for two reasons: (1) due to the complexity of file systems, the number of possible states grows exponentially with the number of system components, a problem known as *state explosion* [39], and (2) state coverage does not provide clear guidance to developers on how to improve their tests. As states are often difficult to predict and accurately represent, developers may not know which specific test cases or scenarios need improvement [127].

Fuzzing [203, 203, 201] uses code coverage as guidance and employs heuristics to prioritize test inputs that increase coverage. However, it lacks guarantees for achieving comprehensive coverage or accessing hard-to-reach code paths.

Automatic test generation [151, 31] creates rule-based syscall workloads to test file system functionality and reliability, typically covering various combinations of syscalls. For example, CrashMonkey [151] exhaustively permutes syscalls within a defined bound to construct operation sequences for testing file system crash consistency. However, focusing solely on syscall combinations is inadequate, as each syscall can have different argument values, leading to different test cases for the same syscall [62].

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

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 [60, 208] 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 [133].

Implementation-level model checking [207, 206] 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 [207, 206] or manually crafting system-specific checkers [206]. Additionally, existing work [207, 206] 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 [206]. Moreover, other model-checking approaches rely on fixed test inputs [60, 206] and lack the versatility to accommodate different input patterns. All model-checking approaches, including Metis, track file system states to guarantee thorough state exploration [39], a feature often lacking in other approaches.

Model checking and fuzzing are orthogonal approaches, each with its own advantages and disadvantages. File system fuzzing [68, 201, 107, 203] 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 kernel instrumentation. Some fuzzing techniques [107, 203] also corrupt metadata to trigger crashes more easily and use library OS [162] 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 [209] 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 [67] (memory errors) and SibylFS [168] (POSIX violations). Current fuzzers configure the tested syscalls but not their arguments [68, 171], 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 [172] and LTP [150] 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 [151, 31, 115] creates rule-based syscall workloads (*e.g.*, opening a file before writing) and employs external checkers (*e.g.*, KASan [67]) or an oracle [151] 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 [151] exclusively detects crash-consistency bugs. We do not include a comparative analysis of testing for other storage systems, such as NVM libraries [58] and data structures [59], 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 [33] uses an extended Hoare logic to define a crash-safe specification and avoid crash-consistency bugs. Yggdrasil [176] constructs file systems that incorporate automated verification for crash correctness. DFSCQ [32] introduces a metadata-prefix specification to specify the properties of `fsync` and `fdatasync` for avoiding application-level bugs. SFSCQ [90] 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 [33, 176, 32] or concurrency [218]), 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 [107]). 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.

3.3 Parallel and Distributed Model Checking

SPIN and Swarm Verification. The SPIN model checker has evolved over time by adopting various strategies to harness parallel computing resources for faster state exploration. The initial effort extended SPIN to leverage multicore processing [83, 81, 54] and distributed memory [118], followed by the development of algorithms to partition the state space for distributed processing [17], implement random-walk state search [178], and parallelize its execution [26, 16], ultimately leading to the invention of SV. Recent research on SV has focused on enhancing its parallel-execution performance using hardware such as FPGAs [38, 159] and GPUs [42], but challenges related to scalability, isolation, and resource management have remained unaddressed until now.

Parallel and Distributed Model Checking. Unlike the diversification technique used in SV, other methods parallelize model checking on multi-core or distributed systems through state-space partitioning and construction [61, 93], as well as using search algorithms such as parallel breadth-first search [15] and parallel depth-first search [14]. There are also non-SPIN-based model checkers that support both intra-node and inter-node parallel execution. Specifically, DiVinE [13] supports both shared and distributed execution models to address the state explosion problem, using multi-threading for shared-memory systems and MPI for distributed environments. Murφ [180] partitions the global state space by hashing each state to a designated worker, uses a parallel BFS algorithm, and employs message passing to exchange states between workers on a multicore machine or a cluster. Eddy [146] improves Murφ's architecture by introducing two threads per node: one for state generation and one for communication, reducing synchronization and context-switching overhead while improving intra-node parallelism.

VTs in SV and CoSV are not coordinated, eliminating communication and synchronization overhead. Model checkers like Murφ incorporate load balancing into their parallel execution [113], whereas SV requires manual effort to configure resource limits and availability on each machine. CoSV enhances SV by adding automatic VT assignment to nodes, dynamically monitoring resource availability to assign VTs accordingly. Additionally, CoSV offers fault isolation and runtime consistency, which are often lacking in those approaches.

Chapter 4

Enhanced File System Testing through Input and Output Coverage

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

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

4.1 Introduction

File systems serve as the backbone of modern storage, supporting numerous applications such as databases, cloud platforms, and local computing devices [133]. Given their critical role, file system bugs pose significant risks to overall system reliability [71, 149], including data loss, data corruption, and system crashes. Consequently, various testing techniques have been developed to detect file system bugs and improve system reliability [203, 127, 151]. File system testing, however, remains a challenge due to the complexity of file systems and their stringent requirements, such as data integrity, fault tolerance, and POSIX compliance [168]. Despite the availability of a number of testing tools, such bugs continue to emerge on a regular basis [107], indicating that existing testing methods are inadequate and there is room for improvement.

Various *coverage metrics* have been proposed based on specific testing approaches. For example, regression testing [172, 150] seeks to achieve functionality coverage, while model checking targets state coverage [127, 182]. *Code coverage* is the most widely used metric in file system testing [107]. However, the effectiveness of code coverage for file systems remains insufficiently studied. It is still unclear whether increased code coverage leads to identifying more bugs. Additionally, even when developers know which code segments are not covered, modifying tests to enhance coverage is a challenge due to the complexity of file system code [60, 8].

Most existing analyses of code coverage effectiveness [69, 92] focus on small user applications rather than large, low-level systems like file systems. Furthermore, no coverage metrics have been specifically designed for file system testing to help developers improve testing and detect more bugs.

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

Computing input and output coverage involves tracing tested syscalls while excluding unrelated noise that is not part of the test workload. Because existing syscall tracers [4] cannot solely focus on tracing file system calls, we designed and implemented **IOCov** to compute input and output coverage for testing tools. We applied IOCov to various file system testing tools, including black-box testing, regression testing, fuzzing, and model checking, uncovering untested input/output partitions in all of them and gaining insights into how they can be improved.

To demonstrate the utility of IOCov, we enhanced the crash consistency testing tool CrashMonkey [151] by 1) significantly improving its input coverage while keeping the rest of the system unchanged, and 2) having it run IOCov. The improvement in input coverage comes from a driver that provides more diverse syscall arguments (*i.e.*, inputs) than the original CrashMonkey. We refer to this new version of CrashMonkey as **CM-IOCov**. We compared CM-IOCov to the original CrashMonkey in terms of the ability to detect crash-consistency bugs in the Btrfs file system [169], and found that CM-IOCov identified 74.1% more test failures (potential bugs) than the unmodified CrashMonkey on the new Linux 6.12 kernel.

In summary, this chapter makes the following contributions:

1. By using xfstests to analyze real bugs, we revealed the limitations of code coverage (it often misses bugs even in covered code) and highlighted the importance of covering diverse syscall inputs and outputs.

2. We formalized *input and output coverage*, allowing us to evaluate and improve file system testing by partitioning input and output spaces, thereby addressing the limitations of code coverage.

3. We designed and implemented IOCov to evaluate the input and output coverage of file system testing tools. We applied IOCov to a number of testing tools, in the process deriving insights for their improvement.

4. We created CM-IOCov to enhance crash-consistency testing (*i.e.*, CrashMonkey). CM-IOCov detects more bugs than CrashMonkey and demonstrates the effectiveness of input coverage in real-world bug detection.

The rest of this chapter is organized as follows. Section 4.2 considers the effectiveness (or lack thereof) of code coverage when it comes to bug detection, and underscores the role inputs and outputs can play here. Section 4.3 defines input/output coverage, and presents the design and implementation of the IOCov

framework. Section 4.4 focuses on the role CM-IOCov plays in improving crash consistency testing. Section 4.6 presents our experimental results, Section 3.1 discusses related work, and Section 4.7 offers our concluding remarks.

4.2 File System Bug Study

This section addresses two questions: (1) whether code coverage is effective for file system testing, and (2) which aspects of testing are crucial for detecting bugs. To answer these questions, we analyzed recent file system bugs which led us to devise *input and output coverage criteria* for file system testing. Unlike previous file system bug studies that focused on bug patterns and classification [133, 212], our study not only examined the effectiveness (or lack thereof) of code coverage in finding bugs, but also identified the key factors that contribute to bug detection.

4.2.1 Code Coverage in FS Testing

A common approach to assessing code coverage effectiveness is *mutation testing*, which involves introducing small faults and checking whether test suites with increased code coverage can detect more faults than those with lower coverage [92, 69]. This method, however, is not applicable to in-kernel file systems, where even small faults can lead to serious OS errors, make the file system unmountable, or damage basic file utilities, preventing us from executing any further tests. As a result, we adopt a different approach to evaluating the correlation between code coverage and bug detection by studying known file system bugs [109, 77]. If covering the buggy part of the code helps a test detect the underlying bug, it suggests that code coverage is effective.

Developers identify and resolve Linux kernel file system bugs by submitting *patches*, which, after review, are merged into the kernel repository [96]. Therefore, analyzing *accepted* patches in the form of Git commits can reveal information about previously buggy file system code [184]. We collected the 100 most recent Git commits [187] from 2022 for each of the two popular Linux file systems, ext4 [144] and Btrfs [169], amounting to 200 commits in total. These were the latest commits available when we began this project. Some commits were not bug fixes; instead they introduced new features, performance optimizations, or maintenance changes [133].

We then applied Lu *et al.*'s taxonomy [133] to identify bug-fix commits, finding 51 ext4 bugs and 19 Btrfs bugs. (The lower count for Btrfs is due to major code

refactoring in December 2022.) These 70 bugs span all four file system bug categories of Lu *et al.*'s taxonomy: 37 semantic, 3 concurrency, 20 memory, and 10 error-code bugs. Next, we ran xfstests, a widely used test suite, on ext4 and Btrfs using Linux kernel v6.0.6, the latest version in which the extracted bugs remained unfixed. We executed all generic and file system-specific tests and used `Gcov` [91] to measure line, function, and branch coverage of the file system sources.

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

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

4.2.2 Keys for Bug Detection

Given the limited effectiveness of code coverage, we further investigated why covering code does not always reveal bugs, and identified key factors for detection. To this end, we analyzed each bug to determine the test cases (including syscalls and their arguments) needed to find it. We observed that many bugs can only be detected when specific syscalls and particular arguments are used. Executing calls with ineffective argument values may cover the code but fail to expose the bug. We refer to bugs that require specific argument values to trigger them as *input bugs*. Moreover, we found that many bugs occur in the exit or error paths of kernel functions, potentially affecting syscall return values and error codes [139, 78]; we refer to these as *output bugs*. Covering syscall return behavior is crucial for

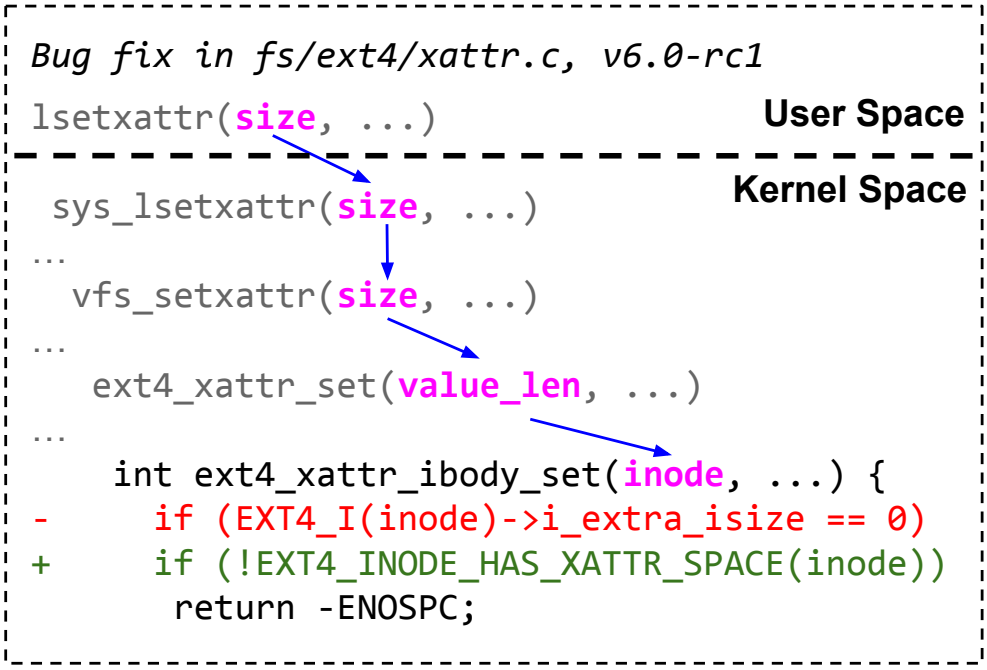


Figure 4.1: An ext4 bug that qualifies as both an input and output bug. The bug was fixed by checking whether the inode has room to store additional xattrs in `ext4_xattr_ibody_set`.

revealing them.

Figure 4.1 shows an ext4 bug [190], fixed in Linux kernel version v6.0-rc1, which qualifies as both an input and an output bug. It is an input bug because it occurred only when `lsetxattr` used the maximum legal `size` argument, causing the minimum offset (`min_offs`) between two block groups to overflow. Although its lines, function, and branches are all covered by xfstests, the test suite failed to detect it. It is also an output bug because it occurs on a function's exit path and affects the behavior of an error code (*i.e.*, `ENOSPC`).

To determine a bug's classification as an input bug, output bug, both, or neither, we analyzed each bug in terms of the inputs required to trigger it and the outputs it can impact. Of the 70 bugs analyzed, we identified 50 as input bugs (71%), 41 as output bugs (59%), and 57 as either input or output bugs (81%). The prevalence of input and output bugs (or both) underscores the necessity of ensuring thorough coverage of inputs and outputs in file system testing. Additionally, of the 37 bugs missed by xfstests despite covering their lines of code, 28 (76%) are input or output

bugs. Among them, 24 (65%) are input bugs that require specific syscall argument values to be triggered. These argument values often involve corner cases, such as out-of-bound reads [21], less-tested inputs such as extended attributes [20, 103], and links [19], and boundary values that trigger overflow [190, 196] or are zero-length [197].

Code-coverage metrics typically do not take into account the diversity of input cases, with lightly tested inputs often following the same execution paths as well-tested ones [188]. Consequently, code coverage does not weight repeatedly executing the same code path with varying inputs [77]. Similarly, bugs in various output cases, such as error codes [139, 119], are not properly evaluated by such metrics.

Thus, we need to consider comprehensive coverage of input types, including syscalls and their arguments, as well as outputs such as return values and error codes. Given the lack of established metrics and tools for measuring input and output coverage in file system testing [112], we propose input and output coverage metrics and present IOCov as a means for evaluating them. In summary, code coverage alone is insufficient for testing, as many bugs rely on specific inputs and outputs that may be missed. Covering syscall inputs and outputs during testing helps address these limitations.

4.3 IOCov Metrics and Framework

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

4.3.1 Input and Output Coverage

Linux provides over 400 system calls, with many specifically related to file systems [12, 188]. Each system call can take multiple arguments, and both the arguments and the outputs can assume arbitrary values from large domains. Consequently, it is infeasible to evaluate whether all possible inputs and outputs are covered by testing. We observed, however, that file system calls exhibit structured input and output patterns, which can be partitioned into categories containing

semantically similar cases. For example, the `open` syscall has a bitwise `flags` argument, where each flag represents a specific behavioral option. Enabling a particular flag triggers behavior tied to a distinct aspect of the syscall. For example, setting `O_CREAT` causes a file to be created if it does not already exist. Instead of analyzing all flag combinations, which would be exponentially complex, we treat each flag independently and check whether it appears in any test input. This reveals how well a testing tool covers the behaviors encoded by `open` flags. This partitioning strategy can also be applied to other inputs and outputs, but different input and output types require different methods.

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

For input coverage, we exclude identifiers and pointers because their values correspond to specific operands or memory addresses and do not represent semantically distinct test cases. We partition numeric arguments using boundary-value analysis [167, 157, 46, 213], selecting powers of 2 as boundaries since they are commonly used in file systems [100]. Each partition is defined as the range between two adjacent boundaries. In the case of categorical arguments, each predefined option corresponds to a unique input partition.

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

Partitioning the input/output space enables us to define coverage based on semantic groupings, eliminating the need to examine every possible value. As such, we define *input coverage* and *output coverage* as metrics that describe the extent to which a testing tool exercises the input and output partitions of file system calls. Input and output coverage offer insights for improvement in both *completeness*, which measures how thoroughly the tool exercises all defined input and output partitions, and *balance*, which measures how evenly test cases are distributed across those partitions. Ideally, a tool should cover as many input

and output partitions as possible to ensure comprehensive coverage and achieve completeness. Moreover, given limited resources, it is important to avoid over-testing certain partitions and under-testing others to ensure balanced coverage.

Achieving both objectives is often challenging in practice. For example, the `open` flag `O_LARGEFILE` is intended for legacy 32-bit systems and is rarely used on modern 64-bit computers. It is therefore difficult to justify testing it as extensively as commonly used flags like `O_RDONLY`. Triggering hard-to-reach outputs such as `ENOMEM` requires a system with limited memory, making such cases more difficult to test.

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

4.3.2 IOCov Architecture

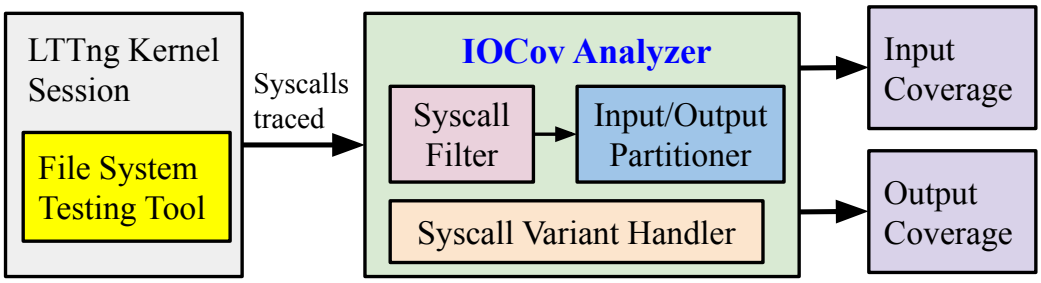


Figure 4.2: IOCov architecture and components.

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

We designed and implemented IOCov to address the limitations of existing tracers and to accurately compute input and output coverage. Figure 4.2 illustrates IOCov's architecture, its core components, and the workflow for computing coverage with a file system testing tool. We leverage LTTng [43] to trace the syscalls. Compared to other tracers such as `strace` [181] (which incurs high overhead), `ftrace` [121] (which may require manual processing for interpretation), and `bpftrace` [24] (which requires custom scripts and may face scalability challenges under high-frequency, multi-threaded workloads), LTTng offers low-overhead, out-of-band tracing with full syscall input/output capture and good scalability [4]. IOCov executes a given file system testing tool within an *LTTng Kernel Session*, allowing all syscalls invoked by the tool, including their inputs and outputs, to be recorded.

The *IOCov Analyzer* then processes the syscalls traced by LTTng. It has three components: the *Syscall Filter*, the *Input/Output Partitioner*, and the *Syscall Variant Handler*. The *Syscall Filter*, described in detail in Section 4.3.3, analyzes raw traces and removes noisy or unrelated syscalls. Once the syscalls relevant for testing are identified, the *Input/Output Partitioner* collects their inputs from `syscall_entry` events and outputs from `syscall_exit` events, determines which partition each input and output value belongs to based on the method described in Section 4.3.1, and counts the occurrences of each input and output partition.

We observed that the testing tool can trigger file system calls that perform equivalent functions. For example, the `openat` syscall serves a similar purpose as `open` but allows one to specify a directory file descriptor for more flexible path resolution. We refer to the original or earliest form of a syscall, such as `open`, as a *base syscall*, and to the extended or modified versions derived from it as *syscall variants*. Since these variants share much of the same kernel implementation as their corresponding base syscalls [188, 163], the *Syscall Variant Handler* groups the base syscall and its variants together and merges their input and output spaces when computing coverage in the IOCov analyzer.

Table 4.1 lists the supported syscalls, along with their arguments and categories as defined in Section 4.3.1. IOCov supports 23 syscalls (nine base and 14 variants) for coverage computation, and also collects additional syscalls (*e.g.*, `close`, `chdir`) to help identify file descriptors and pathnames associated with the test workload. After processing by the *IOCov Analyzer*, a report of the testing tool's input and output coverage is produced using a nested JSON key-value format, capturing the occurrence count of each input and output partition for every syscall supported by IOCov.

Table 4.1: Base syscalls and their variants supported by IOCov, along with the arguments used for input coverage. Each argument is annotated with its type: B (bitmask), N (numeric), and C (categorical).

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

4.3.3 IOCov Filtering Mechanism

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

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

Algorithm 1: Syscall filtering in IOCov to retain only those related to testing.

Input: *trace_file*: trace output of the testing tool
Output: *filtered_syscalls*: test-related traces

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

test mount point. If the syscall is `open` or one of its variants, and both `IsTesting` returns true and the syscall completed successfully, the returned file descriptor is added to `fd_set` to represent file descriptors associated with testing.

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

4.4 CM-IOCov: IOCov for CrashMonkey

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

4.4.1 CM-IOCov Architecture

CrashMonkey [151, 152], which tests file system correctness under crash scenarios, does not actually crash the file system. Instead, it simulates crashes by recording I/O and replaying it up to a persistence point (*e.g.*, after an `fsync()` call). It generates test workloads as short sequences of syscalls, each followed by an explicit persistence operation such as an `fsync` or a global `sync`. It then compares the file system state after a simulated crash to a corresponding oracle, which represents the expected state after a safe unmount, and treats any mismatch as a crash consistency bug.

CM-IOCov improves CrashMonkey's test workload generation while reusing its crash simulation, oracle generation, and state comparison components. By only improving input coverage, we isolate its impact from other potential improvements to clearly observe its effect. Figure 4.3 illustrates the CM-IOCov architecture. The original CrashMonkey workload generator first builds syscall sequences and then fills in the arguments to satisfy operation dependencies (*e.g.*, creating a file before accessing it). CM-IOCov uses the same syscall sequences and dependency resolution strategy for file and directory objects as CrashMonkey, but employs

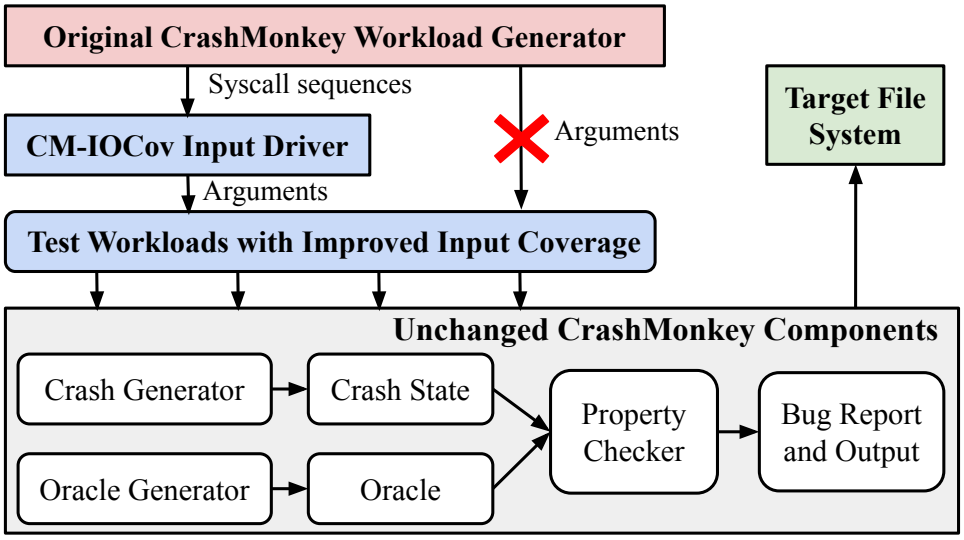


Figure 4.3: CM-IOCov architecture: improves CrashMonkey's test workload generation by introducing an input driver that provides syscall inputs with higher input coverage.

the *CM-IOCov Input Driver* to generate argument values that offer better input coverage for syscalls supported by IOCov.

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

4.4.2 CM-IOCov Input Driver

CM-IOCov's key component is its *Input Driver*, designed to generate more diverse syscall arguments than CrashMonkey, thereby achieving improved input coverage and finding bugs that CrashMonkey misses. Table 4.2 provides a list of syscalls and their inputs where CM-IOCov improves upon CrashMonkey. In particular, for `open()`, CrashMonkey uses fixed values for the `mode` and `flags` arguments,

`0777` and `O_CREAT | O_RDWR` respectively, which always creates a file with full permissions and read-write access for all users, but fails to explore other file creation scenarios. CM-IOCov creates files using a broader combination of modes and flags than CrashMonkey, including various read, write, and execute permission settings for non-owner users and groups, as well as additional file creation flags such as `O_TRUNC`, which truncates a file to zero length if it already exists, and `O_APPEND`, which ensures that all writes are appended to the end of the file. For numeric arguments such as the size in `write`/`fallocate` and the length in `truncate`, CrashMonkey uses a fixed value across all cases. For example, when appending to a file, it always writes 32,768 bytes, regardless of the file system type or underlying disk size.

By contrast, CM-IOCov incorporates multiple input generators tailored to each scenario involving byte arguments, including power-of-two values for aligned cases and $2^n \pm 1$ values to capture unaligned cases and edge conditions near alignment boundaries. This enables exploration of a wider range of numeric byte values, achieving better input coverage than CrashMonkey in cases such as appending to a file, overwriting, or truncating a file. For test cases involving file extension via `write`, CM-IOCov uses two input generators: one for the offset and one for the write size. This ensures coverage of both overlapping writes and writes that extend the file. CM-IOCov also takes the file system's disk size into account when generating byte arguments so that file writes stay within the available space, thereby avoiding `ENOSPC` failures. CM-IOCov also supports additional file system operations present in CrashMonkey that are not shown in Table 4.2, such as direct-IO write and `mmap` write, for which the input driver can improve syscall inputs.

For each argument supported by IOCov, CM-IOCov constructs a value pool containing relevant inputs, such as those produced by the byte argument generators mentioned above, to achieve high input coverage for various test scenarios. Once the syscall sequence is determined, CM-IOCov randomly selects a value from the corresponding argument pool for each argument.

Unlike semantic file system testing, which also examines error cases [168], crash consistency testing requires syscalls to execute successfully in order to verify crash-time properties [33], such as whether a successfully created file persists after a crash. Therefore, CM-IOCov excludes inputs that could cause file system operations to fail, such as invalid flags that prevent file creation or write sizes that exceed the available disk space. Due to the vast number of syscall sequences [151], randomized input selection from value pools enables coverage of diverse input spaces and improves overall input coverage.

4.5 Implementation

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

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

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

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

combinations. A second challenge is that the capability of certain file system testing tools to improve input and output coverage is constrained. For example, CrashMonkey mandates that syscalls succeed, which prevents it from exploring outputs in the form of error codes.

Therefore, the effectiveness of testing improvements depends on the nature of the testing goal. Output coverage is important for semantic testing [168, 149], but it is not essential for crash consistency testing [151, 115, 176]. Additionally, different testing tools may focus on specific input or output partitions, such as writing many small files or a single large file [150]; thus, aligning input/output coverage with test goals still requires domain knowledge and manual effort. Nevertheless, input and output coverage, along with IOCov, provide an effective methodology for developers to use to evaluate and enhance testing more easily than traditional code coverage. We demonstrated the utility of our approach by using it to develop CM-IOCov and thereby improving existing crash consistency testing.

4.6 Evaluation

In this section, we address the following questions: **(1)** What are the overhead and performance impacts of applying IOCov to file system testing tools (§4.6.1)? **(2)** How accurately does IOCov compute input and output coverage in file system testing (§4.6.2)? **(3)** How can IOCov be utilized to assess the coverage of existing testing tools and provide developers with insights for improvement (§4.6.3)? **(4)** Can CM-IOCov improve bug detection and discover bugs that the original CrashMonkey fails to detect (§4.6.4)?

Experimental setup We conducted all IOCov experiments on two virtual machines (VMs), each equipped with 8 CPU cores and 128GB of RAM. Both VMs ran Ubuntu 22.04 with Linux kernel version 5.19.6. We ran all CM-IOCov and CrashMonkey experiments on two additional VMs configured identically to the IOCov machines, except that one used Linux kernel version 5.6 (the latest version supported by the original CrashMonkey) and the other used version 6.12, which is the kernel we migrated the system onto. Each VM was equipped with a 1TB disk partition used to store all generated executables and log files. To evaluate IOCov, we employed four file system testing tools: xfstests [172], Metis [127], Syzkaller [68], and CrashMonkey [151]. These tools were selected as representatives of different testing techniques: regression testing, model checking, fuzzing, and automatic test generation, respectively. As the tools vary in design and capa-

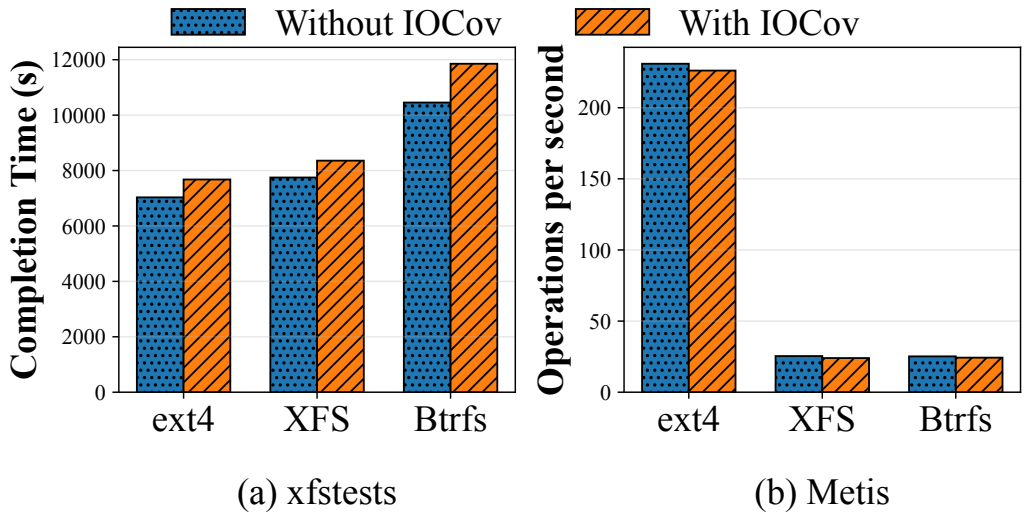


Figure 4.4: Performance overhead of IOCov on xfstests and Metis, measured using completion time and operation rate, respectively.

bilities, we selected appropriate tools for each evaluation task, while measuring input and output coverage across all of them.

4.6.1 IOCov Overhead

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

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

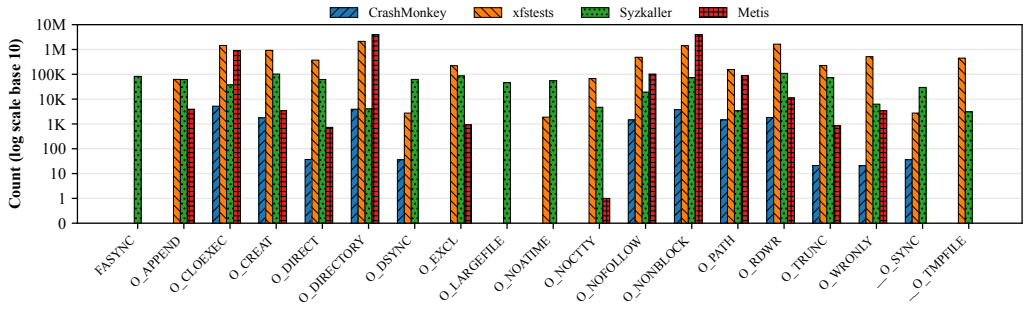


Figure 4.5: Log_{10}-scaled input coverage counts (y-axis) for each `open` flag (x-axis), measured across CrashMonkey, xfstests, Syzkaller, and Metis.

dynamically at runtime and can run for extended periods. Therefore, overhead is measured as the difference in Metis's speed (operations per second) when run with and without IOCov.

Figure 4.4 compares the performance of xfstests and Metis with and without IOCov. Using three file systems (ext4 [144], XFS [177], and Btrfs [169]), we executed all of xfstests' generic tests and ran Metis for one hour. With IOCov, the overhead from LTTng increased xfstests' completion time by 7.8–13.4%. For Metis, more operations per second indicates better performance; with IOCov, the operation rate dropped by 2.1–5.3%. Metis runs faster on ext4 than on XFS and Btrfs because it uses the minimum allowed device size: 2 MiB for ext4 compared to 16 MiB for the others. Metis takes longer to save and process states for larger devices. For post-tracing analysis, IOCov took 22.5 minutes to extract coverage information for xfstests applied to ext4 and 17.4 minutes for Metis.

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

4.6.2 IOCov Accuracy

It is important that IOCov accurately filter syscalls related to file system testing for coverage computation, a property we refer to as its *accuracy*. To evaluate accuracy, we compared its reported coverage with the verified inputs and outputs exercised during testing, which served as the ground truth. Many file system testing tools do

not log the syscalls they execute for testing (*e.g.*, xfstests), or retain only incomplete information (*e.g.*, CrashMonkey). As such, we applied Metis to this task, as it records file system calls along with their inputs and outputs, capturing only those relevant to testing.

This accuracy experiment compared the coverage reported by IOCov for Metis against the inputs and outputs recorded in Metis logs. We ran Metis for one hour on ext4 while using LTTng to trace syscalls. We then computed input/output coverage using IOCov and measured the expected coverage based on Metis logs. For the comparison, we counted the file system calls captured by IOCov (computed coverage) and those recorded in the Metis logs (expected coverage). We found that both recorded similar counts for most file system syscalls. IOCov recorded 39,641 calls to `write`, 523,152 to `truncate`, 29,280 to `mkdir`, and 54,486 to `chmod` while Metis reported 41,935 calls, 523,047 calls, 29,261 calls, and 54,475 calls, respectively. The highest error rate among the syscalls was for `write` at 5.47%, which is acceptable for understanding the testing tool's input and output coverage. The mismatch comes from slight LTTng instrumentation limitations or timing discrepancies, which may cause certain syscalls to be missed or logged inaccurately compared to ground-truth execution.

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

4.6.3 IOCov Use Cases in Testing

We used IOCov to measure input and output coverage for different types of testing tools in order to evaluate them and deliver insights on how they can be im-

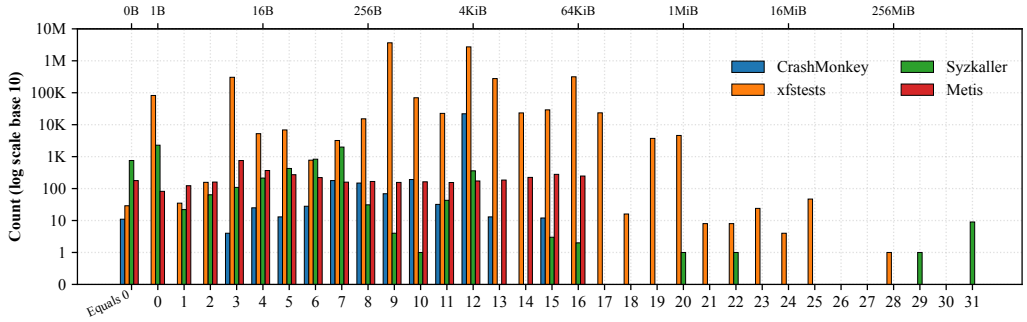


Figure 4.6: Log_{10}-scaled input coverage counts (y-axis) for `write` sizes (in bytes), measured across CrashMonkey, xfstests, Syzkaller, and metis. The x-axis shows powers of 2 for write sizes, with a special entry labeled "Equals 0" for writes of size zero.

proved. Our evaluation covers four tools: CrashMonkey [151], xfstests [172], Syzkaller [68], and Metis [127]. These tools vary in their characteristics and runtime behavior. CrashMonkey generates grouped workloads based on test length; xfstests runs a fixed set of workloads, with total duration depending on the selected test groups; Syzkaller and Metis continuously generate workloads and can run for extended periods. To ensure fairness, we selected workloads for CrashMonkey (including `seq-1` and other default workloads) and xfstests (the `quick` group) that complete within one hour. We then ran Syzkaller and Metis for the same duration. Each tool was evaluated using the same underlying file system, ext4. Specifically, Metis allows flexible configuration of syscall input distributions; however, we used its default settings without additional customization. Although Syzkaller is not a dedicated file system fuzzer, it can trigger file system bugs through relevant syscalls, so we restricted it to generate only file system-related calls. While IOCov is capable of generating comprehensive coverage data for all supported syscall inputs and outputs, due to space constraints we report and analyze only a representative subset.

Figure 4.5 shows input coverage for `open` flags across these four tools. The x-axis shows individual flags, each representing a single bit, and the y-axis (log_{10} scale) indicates how frequently each flag was exercised by the testing tool. Input coverage for `open` flags is measured over all input partitions, *i.e.*, the individual flags that compose the bitmask. Among the four tools, Syzkaller achieves the most thorough input coverage for `open` flags—exercising all flags and being the only tool that covers both `FASYNC` and `O_LARGEFILE`. Although Metis and xfstests

cover many commonly tested flags, they miss certain ones that are also worth testing. For example, `O_LARGEFILE`, which enables support for large files, may still expose bugs [195] in modern systems. CrashMonkey covers the fewest flags and even misses common ones such as `O_APPEND`, indicating a need for improved input coverage (which led to the development of CM-IOCov). Furthermore, our results show that some flags are exercised millions of times, while others are never tested, revealing a significant imbalance in test coverage. Such information can inform the reallocation of test efforts to under-tested cases.

Figure 4.6 shows the input coverage for the `write` size argument (count in bytes), partitioned by boundary values based on powers of 2. The x-axis represents the $\log_2$ of the `write` size, while the $x2$-axis shows the corresponding actual size. For example, $x = 8$ represents all sizes from 2^8 to $2^9 - 1$ (*i.e.*, 256–511 bytes), with the corresponding $x2$-axis value shown as 256 B. The y-axis ($\log_{10}$) shows how many times each x-axis bucket was tested by a given tool. As shown in Figure 4.6, all four tools prioritize testing small sizes (less than 4 KiB) and lack coverage for larger sizes. While Syzkaller and xfstests covered a broader range of `write` size partitions than Metis and CrashMonkey, all four tools completely missed some partitions. Similarly, all tools showed uneven testing of `write` sizes. For instance, xfstests exercised some size partitions millions of times, while others were not tested at all. This coverage information provides developers with direct insights into how to improve test cases and address untested scenarios.

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

4.6.4 CM-IOCov Bug Finding

IOCov's ultimate objective is to enhance existing test tools to uncover file system bugs that the originals miss. We developed CM-IOCov to enhance CrashMonkey's crash consistency testing by leveraging IOCov's coverage reports, and evaluated whether it finds more bugs than the original CrashMonkey on Linux kernel versions 5.6 and 6.12. Linux 5.6 is the latest kernel that the unmodified CrashMonkey supports, but bugs in this version may already be fixed in newer kernels. We ported both CrashMonkey and CM-IOCov to Linux 6.12 by updating their code and kernel API usage, allowing them to run on the newer kernel and discover more recent bugs. We evaluated the Btrfs file system, which is the main file system targeted by CrashMonkey [151], on both kernels.

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

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

For the 6.12 kernel, we ran 391,134 test workloads using both CM-IOCov and CrashMonkey. During that experiment, CM-IOCov reported 390 failures, compared to 224 found by CrashMonkey. CM-IOCov found 323 failures missed by CrashMonkey, while CrashMonkey found only 157 missed by CM-IOCov, again demonstrating CM-IOCov's superior bug detection ability. Table 4.3 presents five representative Btrfs bugs identified by CM-IOCov but missed by CrashMonkey, along with their consequences and the syscalls that triggered them. Across the bugs in Table 4.3 as well as the failures uniquely identified by IOCov, most of the involved syscalls benefited from CM-IOCov's improved input generation; these syscalls are underlined in the table. This highlights the importance of input coverage for bug detection. The crash-consistency bugs found by CM-IOCov have serious consequences, such as allocated blocks being lost despite explicit persistence via `fsync()`, file content or hard link counts not being persisted, and files missing after a crash. Since these bugs were found on Linux kernel 6.12, they are likely to be real and still present. We are actively investigating them and plan to report them to Btrfs developers with detailed diagnostic information. While CrashMonkey did detect some failures that CM-IOCov missed under the same workload count, its inputs are a subset of CM-IOCov's. Therefore, with enough workloads, we believe CM-IOCov can also reveal those failures.

4.7 Chapter Conclusion

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

Table 4.2: Comparison of syscall inputs generated for testing in CrashMonkey and CM-IOCov, showing how CM-IOCov improves input coverage over CrashMonkey.

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

Table 4.3: Crash consistency bugs in Btrfs, their consequences, and the triggering call sequences. These bugs were detected by CM-IOCov but missed by CrashMonkey on Linux 6.12. Underlines indicate inputs improved by CM-IOCov over CrashMonkey.

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

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 [107] and enhancing overall system reliability [71], as file-system bugs can have severe consequences [133, 212]. Effective testing of file systems is challenging, however, due to their inherent complexity [8], including many corner cases [207], myriad functionalities [22], and consistency requirements (*e.g.*, crash consistency [161, 176]). Developers have created various testing technologies [206, 172, 151] for file systems, but new bugs (both in-kernel and non-kernel) continue to emerge on a regular basis [107, 203, 106].

To expose a file-system bug, a testing tool must execute a particular system call using specific inputs on a given file-system state [133, 207, 128, 129]. For example, identifying a well-known Ext4 bug [116] 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 [128, 129, 23] 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 [107, 203, 151, 31]. For example, handwritten regression tools like xfstests [172] can achieve good test coverage of specific file-system features [150, 8], but do not comprehensively cover syscall inputs; similarly, fuzzing techniques (*e.g.*, Syzkaller [68]) are designed to maximize code—not input—coverage [102].

Both the input and state spaces of file systems are too vast to be completely explored and tested [60, 29], so it is better to leverage finite resources by focusing on the most pertinent inputs and states [128, 129, 208, 206]. 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 [168]. Existing testing technologies also lack the versatility to test specific inputs and states [172, 151, 68]. Thus, new testing tools and techniques are needed [128, 129, 133] to avoid under-testing (which could miss potential bugs) or over-testing (which wastes resources that may be better deployed elsewhere).

This chapter 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 [70]. 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 [79]).

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 [79]. 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 [40, 79] file systems. It checks file-system implementations directly, eliminating the need to build a formal model [154]. To manage large file-system input and state spaces, Metis enables parallel and distributed exploration [86] 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 [107], that require kernel instrumentation and utilities. Nevertheless, the quality of the reference file system is pivotal for assessing the behavior of other file systems [70]. 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 [182, 206]. 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 [172, 151, 68]. 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,

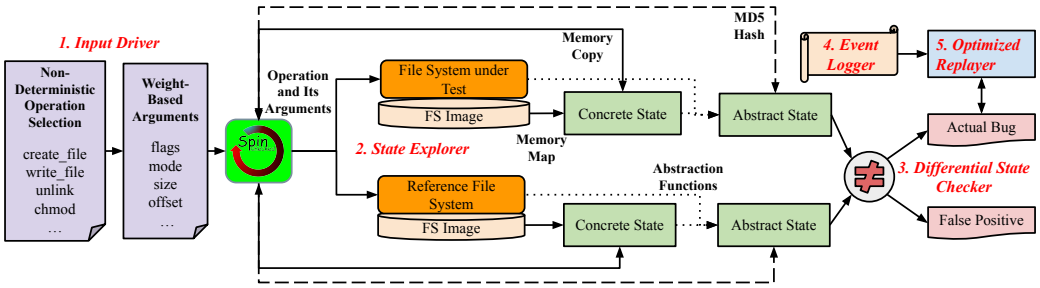


Figure 5.1: Metis architecture and components. From left to right, Metis generates syscalls and their arguments that are executed by both file systems, determines resulting states, and checks for discrepancies between states. The Logger records all the operations for convenient bug replay by the Replayer. 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 chapter 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 [149, 23] 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 [168, 31, 151]. 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 [40, 191]. The model is typically a state machine, and the properties, usually expressed in temporal logic, are checked using state-space exploration [39]; here, each state represents a snapshot of the system under investigation. To automate this process, model checkers (such as SPIN [79]) 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 [207, 206]. Thus, recent work on implementation-level model checking [207, 206] seeks to check the implementation directly (without a model). Such approaches [206] 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 [207, 206]. 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 [62, 199]. Thoroughly testing file system inputs is challenging. While file-system–related syscalls represent only a subset of all Linux syscalls [188, 12], each syscall has multiple arguments, and the potential value range for these arguments is vast [188, 128, 129]. 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 [68, 99]. Instead, a sensible approach [112, 128, 129] is to *segment* a large input space into multiple, disjoint input partitions—called *input space partitioning* [194, 128, 129, 101]. How much a testing tool examines input partitions is called *input coverage* [76, 189, 112]. 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 [128, 12, 188]: (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 [100]). 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 [182, 60]. Comprehensive state exploration is important as certain bugs manifest exclusively under specific states [133, 116, 190]. Numerous file system states can be explored when some existing testing approaches [172, 151] 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 [206]. 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 [60]. 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 [40]. Conversely, an overly narrow definition can skip key states and potentially miss defects [30]. 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 [79] 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 [40]. 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 [86].

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 [79] 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 [70] (*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 [12, 128]. 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 [100], 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 [151], it is used infrequently for POSIX compliance [168]. Due to disk I/O's slow speed, many tests focus on small write sizes [31]. However, testing larger sizes can uncover size-specific bugs [190, 168]. 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 [39]. Metis supports depth-first search (DFS) as the main search algorithm.

The State Explorer relies on the SPIN model checker [79] 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 [30]. 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 [144]) 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 [41, 202], VM snapshotting [108, 114], and LightVM [140]. None of these approaches were effective due to their functional deficiencies or inefficient performance. For those reasons, we adopted the approach presented in [182] 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 [31], such as kernel crashes. We thus needed a dedicated checker to identify cases where file systems fail silently [107] (*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 [151, 168]: they are difficult to adapt to continuously-evolving file systems.

File systems vary considerably in terms of their developmental stages [210, 133]: mature file systems are typically more stable than new, emerging, or less popular ones [133]. Yet many still share common (POSIX) features and data-integrity requirements. Therefore, we rely on a *differential testing* approach [145], 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 imple-

mentations. 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 [211, 107] 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 [151, 107] 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 [79]. 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 [60]. Exploring this space using a single Metis process (called a *verification task*, or VT) requires significant time.

To parallelize the state-space exploration [83] we use Swarm verification [86], 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 [86], 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 [86]. 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 [79] 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 [70, 145], 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 [11], 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 [33]. 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` [73]) 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 [151, 115], Metis cannot find bugs that arise solely during system crashes. We plan to provide the option of invoking utili-

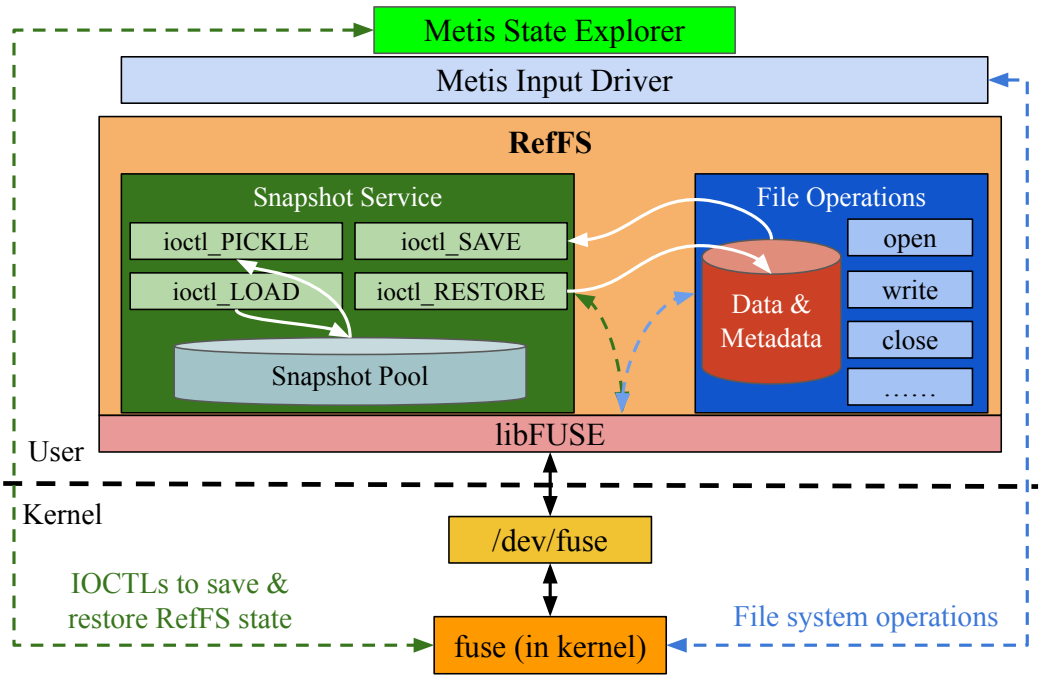


Figure 5.2: RefFS architecture and its interaction with Metis and kernel space. RefFS supports standard POSIX operations and provides snapshot services with a snapshot pool and four new APIs.

ties such as `fsck` [160] 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 [201]). While Metis's replayer assists in reproducing bugs, another limitation is Metis's inability to precisely identify the root cause of detected state discrepancies within the code [170].

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 [144]. 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 [176, 33], 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 [10]. 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 [169] and ZFS [22], 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.

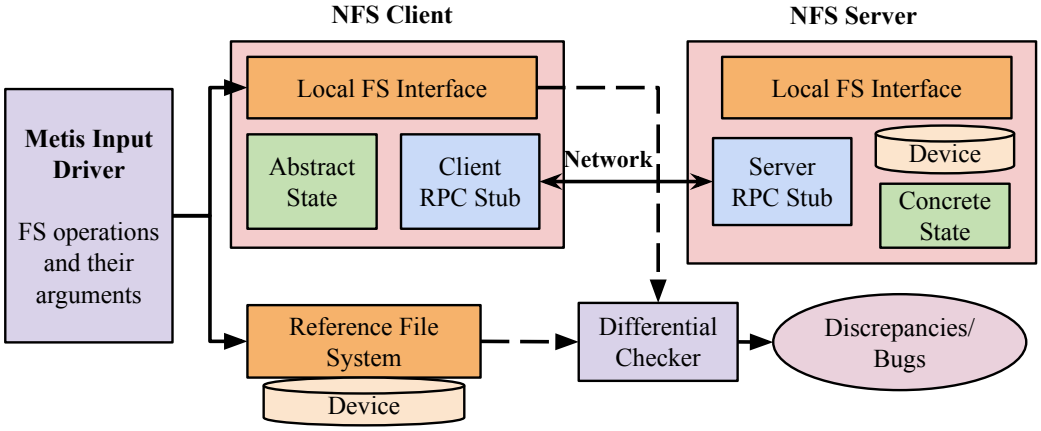


Figure 5.3: The structure of model checking NFS with Metis. We set up one client and one server on the same machine to simplify SPIN's management of concrete and abstract states. The local file system type used by NFS should be identical to the reference file system. Similar to Metis for other file systems,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

5.5.1 The Architecture of Checking NFS

Distributed file systems (DFSs) are another important category of file systems that need thorough checking. However, adapting local file system testing techniques for DFSs is difficult due to factors [183] like network communication, load redistribution, data replication, distributed concurrency, and scalability, which are not notable concerns in local file systems [89]. 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 [127]. As a classic example of a distributed file system, NFS (Network File System) [174] 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 [51] and NFS-Ganesha [156].

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 [41] 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.

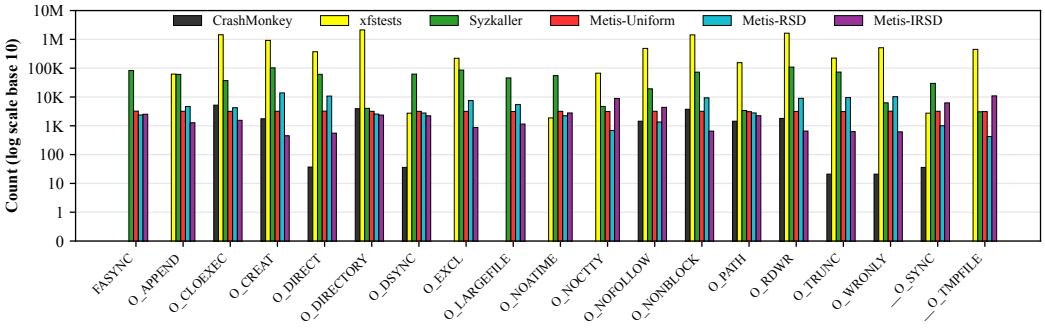


Figure 5.4: Input coverage counts (log_{10}, y-axis) of `open` flags (x-axis) for CrashMonkey, xfstests, Syzkaller, and Metis with 3 different weight distributions.

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 [68] on both dimensions.

Comparison with existing testing tools We selected three testing tools, each representing a unique technique: CrashMonkey [151] for automatic test generation, xfstests [172] for (hand-written) regression testing, and Syzkaller [68] 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 [153].

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 [128] 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 [166]), and IRSD (Inverse Rank-Size Distribution [158]). 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 [195]. 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 millions of times (*e.g.*, `O_DIRECTORY`) 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).

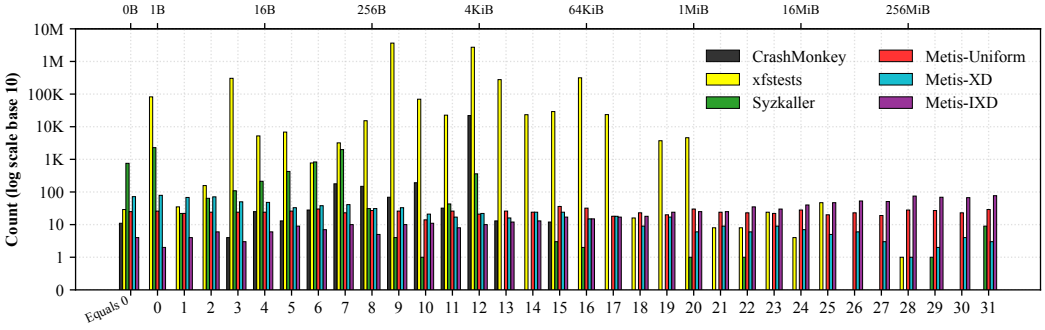


Figure 5.5: Input coverage (counts, log_{10}, y-axis) of `write` size (in bytes) for CrashMonkey, xfstests, Syzkaller, and Metis with three different weight distributions. The x-axis denotes the power of 2 of the write size (shown as x2-axis). Note a special "Equals 0" x-axis value for writes of size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

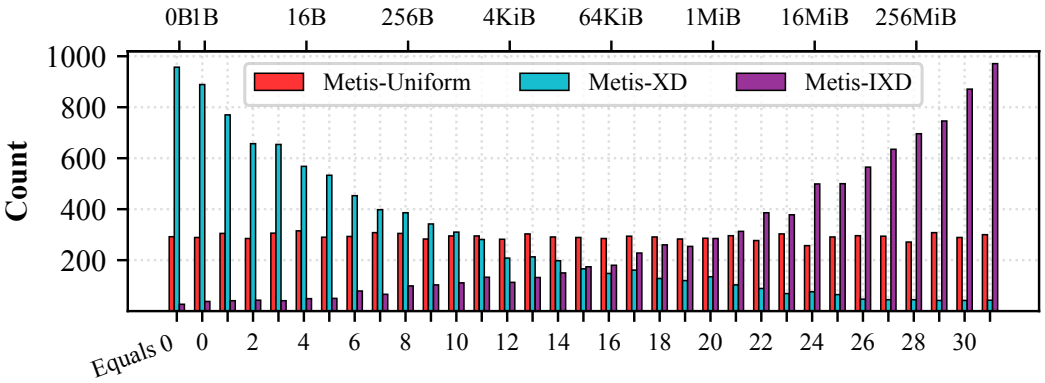
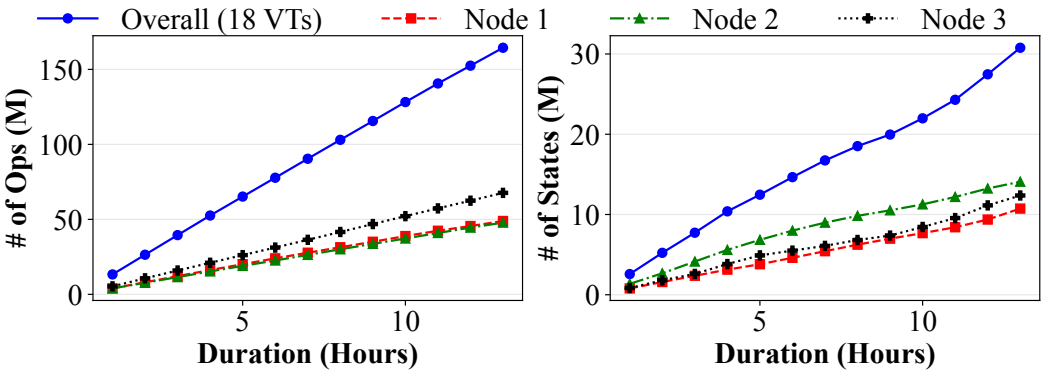


Figure 5.6: Input coverage of `write` size (in bytes) for Metis-Uniform, Metis-XD, and Metis-IXD, each running for 4 hours. The x-axis and $x2$-axis here are the same as in Figure 5.5, but the y-axis shows counts on a linear scale. As observed, extended run times produce distributions that align more closely with the intended distribution, compared to the shorter experiment shown in Figure 5.5.

ing 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× 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× 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%. For brevity, 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× 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 states

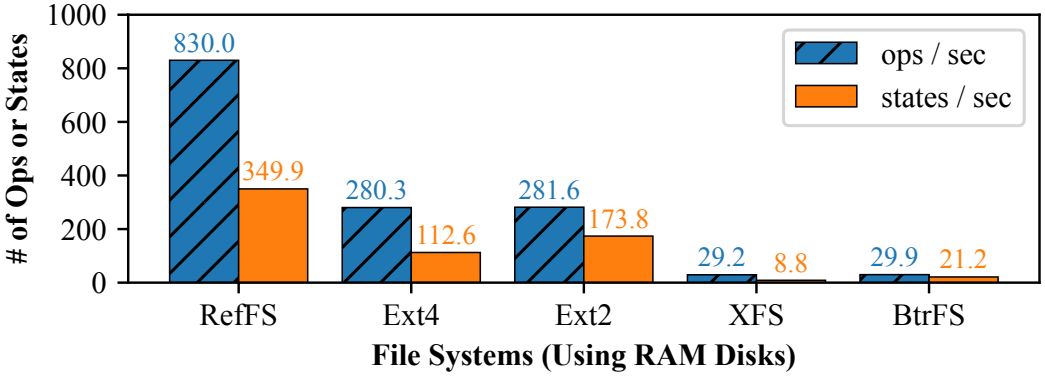


Figure 5.8: Performance comparison between RefFS and other mature file systems while being checked by Metis. The *y*-axis applies to both ops/sec and states/sec.

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 so that we also could use it to debug RefFS. While we used xfstests to find certain

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.

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. This indicates that xfstests can produce false negatives when testing RefFS and provides limited utility for guiding bug reduction efforts. 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 [98], BtrFS [169], F2FS [117], JFFS2 [198], JFS [95], NILFS2 [44], XFS [177], and two persistent memory file systems: NOVA [200], and PMFS [50], 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 [172] or Syzkaller [68]. Metis identified an F2FS bug that was not detected by Hydra [107]. We also checked file systems (*e.g.*, BetrFS) that are not currently supported by Hydra [107].

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

FS Testing Approach	**Input**	**FS effort**	**Ops effort**	**ST**	**CC**	**BD**
Metis: this work	👍👍👍👍	👎	👎	✔	✘	Behavioral discrepancies
Traditional Model Checking: CVFS [60], CREFS [208]	👍	👎👎👎👎	👎👎👎👎	✔	✘	User-specified assertions
Implementation-level Model Checking: FiSC [207], eXplode [206]	👍	👎👎👎	👎👎👎	✔	✘	User-written checkers
Fuzzing: Syzkaller [68], Hydra [107]	👍👍👍	👎👎👎	👎	✘	✔	External checkers
Regression Testing: xfstests [172], LTP [150]	👍	👎👎👎	👎👎👎👎	✘	✘	Preset expected outcome
Automatic Test Generation: CrashMonkey [151], Dogfood [31]	👍👍👍	👎	👎👎👎	✘	✘	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.

deterministic and five were nondeterministic. Deterministic bugs are those easily 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 Chapter 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 chapter, 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× 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).

Chapter 6

CoSV: Containerized Swarm Verification for Scalable and Fault-Isolated Model Checking

Swarm Verification (SV) uses multiple diversified Verification Tasks (VTs) to parallelize SPIN-based model checking. SV, however, suffers from several limitations, including complex deployment, interference among VTs, and resource-management difficulties. We present *Containerized Swarm Verification* (CoSV), where each VT runs in a self-contained, isolated container managed by an orchestrator to handle deployment, scaling, and life-cycle automation. CoSV follows a Controller-Worker architecture, with a single controller node and multiple worker nodes, and can be easily scaled to both cloud and on-premise compute environments. To apply CoSV to a new model or scale the number of VTs, one only needs to configure dependencies (such as external libraries and tools) and specify the resource requirements for each VT, both of which can be accomplished in a single step. We evaluated CoSV on two SPIN-based models: the Metis file system model checker and the Dining Philosophers problem. Our results show that CoSV's performance compares favorably with standard SV. We also demonstrate how CoSV effectively isolates faults among VTs and facilitates scalable deployment of VTs on multi-core servers and in hybrid cloud environments.

6.1 Introduction

Swarm Verification (SV) [84, 86] is a powerful technique that generates multiple parallel *verification tasks* (VTs) to collectively verify complex software systems and efficiently explore large state spaces. SV employs *diversification techniques* to assign varying search parameters (*e.g.*, hash polynomials, random seeds) to the VTs generated by the SPIN model checker [79]. As such, VTs are likely to explore different portions of the state space, thereby collectively increasing state-space coverage.

VTs can be executed in parallel across multiple CPU cores and machines, enhancing efficiency and scalability. However, using SV presents several challenges. For example, SV pre-generates VTs based on a static configuration file and relies on `ssh` to distribute them to remote machines [82]. This approach poses challenges in deploying VTs efficiently across heterogeneous and dynamically changing environments, such as a hybrid cloud with fluctuating server availability [105].[1] Further, VTs running on the same machine lack isolation in terms of their execution environment and resources, leading to potential interference and resource contention. Additionally, VTs operate without a centralized system to monitor their status and coordinate their deployment and resource usage. This results in wasted resources and hinders the identification of property violations within certain VTs [85]. Finally, when SV is scaled to more machines, each machine running VTs must be individually configured with dependencies such as required libraries, tools, and environments. This reduces scalability and adds deployment overhead.

Our contributions. This chapter presents *Containerized Swarm Verification* (CoSV), in which each VT runs in an isolated container—a self-contained environment that includes the application and its dependencies—and is managed by an orchestrator (Kubernetes) that automates the deployment, management, scaling, and operation of containerized VTs. CoSV uses a controller-worker architecture, where a single controller node (machine) manages multiple worker nodes that run all VTs as containers. CoSV packages all of the dependencies needed to containerize a VT into a single step, enabling easy and scalable deployment to worker nodes. This also simplifies adapting CoSV to a new model, as dependencies and VT resource limits only need to be configured once during setup. Once this step is done, CoSV streamlines the packaging, deployment, and management of all

[1] A *hybrid cloud* is mixed computing environment made up of public and private clouds, including on-premises data centers or "edge" locations.

VTs without additional engineering, eliminating the need to configure environments and dependencies on each machine. This efficiency benefits all SPIN-based models that utilize SV.

Additionally, containerized VTs ensure consistent execution across different systems and environments without compatibility issues. This guarantees deterministic and repeatable behavior, independent of external factors such as various runtime configurations. Such an approach aids in the scalable deployment of VTs and in being able to consistently reproduce bugs during model checking by eliminating environment-related variations (*e.g.*, dependency configurations, machine architecture, operating system). CoSV also provides *fault and resource isolation*, preventing runtime issues in one VT from impacting others through fault propagation or resource contention.

We designed and implemented two versions of CoSV: CoSV-Docker and CoSV-Kata, using Docker [47] and Kata Containers [104], respectively, as the underlying container technologies. Different containers offer varying levels of resource and fault isolation, allowing them to be selected based on the needs of different model checking applications. Docker provides higher performance with lightweight isolation, whereas Kata Containers offer stronger isolation through hardware virtualization at the cost of reduced performance. Therefore, CoSV-Docker is suitable for most model-checking scenarios (*e.g.*, Dining Philosophers) that are unlikely to trigger operating system kernel-level failures, whereas CoSV-Kata is more appropriate for system software model checking (*e.g.*, file system verification), where kernel errors could halt all VTs on the same host. We applied CoSV to two representative model-checking tasks: Metis file system model checking [127] and the Dining Philosophers problem [45].

In sum, this chapter makes the following contributions:

1. We designed CoSV to leverage containers and Kubernetes, offering multiple advantages over SV, including improved scalability, fault isolation among VTs, simplified deployment and management, a consistent runtime environment, and more efficient resource usage. Section 6.3.3 considers the advantages of CoSV over SV.

2. We implemented two variants of CoSV using different container software: CoSV-Docker and CoSV-Kata. We applied these variants to two SPIN-based models: the Metis file-system model checker [127] and the Dining Philosophers problem [45].

3. We compared CoSV's performance against standard SV using multiple met-

rics and evaluated its fault-isolation capabilities under two common faults: memory leaks and kernel crashes. We also demonstrated how CoSV facilitates the deployment of VTs across multiple multi-core servers and hybrid cloud environments.

The rest of the chapter is organized as follows: Section 6.2 provides background on Swarm Verification, containerization, and the Metis file system model checker. Section 6.3 describes the architecture of CoSV, including its components, container selection rationale, deployment process, and key advantages and limitations. Section 6.4 presents our experimental results and compares CoSV (including both CoSV-Docker and CoSV-Kata) with SV on the two models in terms of deployment, performance, and fault isolation. Section 6.5 offers our concluding remarks.

6.2 Background and Motivation

This section provides background on three areas relevant to this chapter: Swarm Verification (SV), containerization, and the Metis platform for file system model checking.

6.2.1 Swarm Verification

Given a system model M and a temporal logic property φ, *Model Checking* involves systematically exploring M's state space to determine whether M satisfies φ [39]. SPIN [87] is a widely used model checker for verifying software systems using the Promela (Process Meta Language) modeling language [80], with support for embedded C code to simulate low-level system behaviors, such as those found in file systems [70, 208]. Model checking faces the challenge of state explosion [40], where the state space grows exponentially with the number of system components, making it difficult for model checkers like SPIN to efficiently explore the entire state space and complete verification in a reasonable amount of time. Swarm Verification (SV), an enhancement technique for SPIN and other model checkers, addresses large state spaces by generating multiple parallel VTs, each exploring only a fraction of the space.

The core techniques used by SV to generate VTs are *diversification* and *randomization* [84, 86]. A key diversification technique involves using different hash polynomials per VT to generate unique hash collisions for each VT. Variations

in hash collisions cause each VT to prune state exploration at different points by avoiding revisited states, resulting in distinct exploration patterns. Randomization affects scheduling, parameter selection, and state transitions, and thus each VT follows a different search strategy determined by its random seed. In SV, each VT consists of a series of commands, each of which specifies an executable verifier called `pan` [79], equipped with different arguments defining the search and diversification strategies for the run. An example command is `./pan1 -k1 -w30 -m10000 -h93 -RS2745`, where `pan1` indicates the specific verifier being used, as SV can generate multiple verifiers based on different compilation flags specified in the SV configuration file [82]. The trailing arguments configure the number of hash functions (`-k`), the size of hash table entries (`-w`), the maximum search depth (`-m`), the hash function selection (`-h`), and the seed for the random number generator (`-RS`). Although these techniques assist VTs in exploring different portions of the state space, their deployment, distribution, and management remain insufficiently studied.

In addition to specifying compilation flags for generating different verifiers, the SV configuration file also defines the number of VTs to be launched and how they are to be distributed across multiple machines. For example, specifying `3 remote1:4 remote2:5` indicates generating and executing 3 VTs on the local machine, 4 VTs on remote machine 1, and 5 VTs on remote machine 2, respectively, which requires password-less `ssh` access to all remote machines. This method, however, limits the scalability and flexibility of VT deployment, as SV pre-generates VTs based on a static configuration file. When additional nodes or CPU cores become available, in SV one must update scripts manually to regenerate the VTs and reconnect via `ssh` to the new remote machines.

6.2.2 Metis File System Model Checking

Metis [127] is a SPIN-based implementation-level model-checking framework for Linux kernel file systems that has already discovered 15 bugs. We use Metis as an example to demonstrate CoSV. Metis does not build or analyze abstract models; instead, it directly mutates and explores file system states through file system calls and state representations. Metis leverages SV to scale state exploration across multiple machines, but its scalability and practicality are limited by the design of the standard SV. First, running Metis requires numerous dependencies, including file-system utilities and libraries, making manual distribution of VTs across multiple machines cumbersome. Second, deploying VTs on remote machines relies on `ssh` and `scp` for file transfer and remote execution. This approach, however,

lacks fault tolerance (*e.g.*, to network failures [110]) and makes it difficult to incrementally add or remove VTs in a dynamic environment. Third, VTs on the same machine run as collocated processes without isolation. Metis often triggers different types of kernel bugs, causing VTs on the same machine to interfere with each other [127, 182]. For example, a memory leak may cause the VT that triggers it to consume excessive memory, leading to resource contention with other VTs. A more severe bug can trigger a kernel crash or hang, disrupting all VTs on the same machine.

6.2.3 Containerization

Containerization encapsulates an application and its dependencies into an isolated, self-contained *container*, offering a more lightweight virtualization solution than traditional virtual machines [18]. Central to containerization are tools like Docker [47], a platform that builds and runs containers, and Kubernetes [111], a system that automates the deployment and management of containerized applications.

In a non-containerized environment like standard SV, applications such as VTs are executed using the shared resources and libraries of the host machine, all running under the same OS kernel. Containers generally enable applications to be packaged with their dependencies and run in isolated environments with dedicated resource allocations (*e.g.*, CPU, memory, disk). Container technologies vary in the extent of isolation they provide at the OS and kernel levels. Shared-kernel containers (*e.g.*, Docker) share the host OS kernel, making them lightweight with fast startup and low overhead; a kernel-level error, however, can affect all containers on the same host. In contrast, sandboxed containers (*e.g.*, Kata Containers [104]) run each container in a lightweight virtual machine (VM) with its own kernel, providing stronger isolation so that a kernel-level error in one container does not affect others on the same host, but at the cost of higher overhead. Another benefit of container isolation is the consistent runtime environment it provides. This consistency is important for model checking, which can expose nondeterministic bugs that may not appear reliably across executions, such as race conditions that depend on specific environmental contexts [127, 48]. Once such bugs are identified, containerized environments facilitate more consistent reproduction and help analyze the bugs in a controlled setting [186].

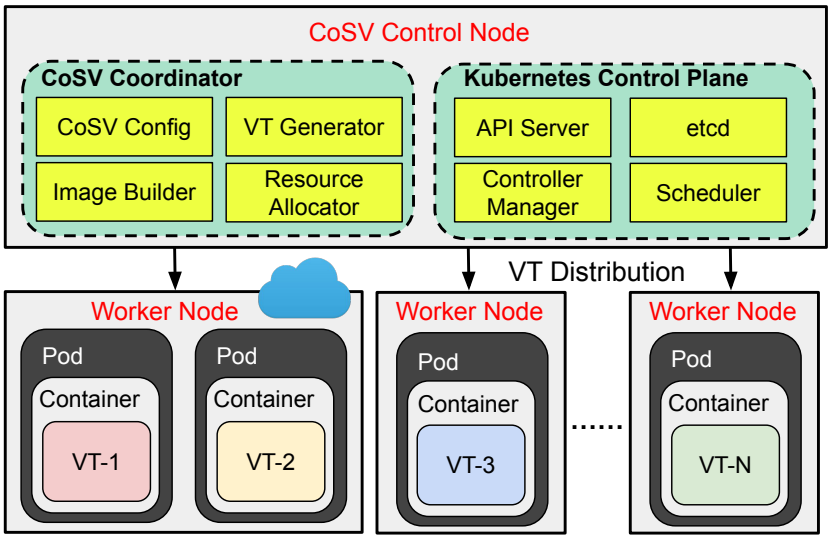


Figure 6.1: CoSV architecture and components.

6.3 CoSV Architecture

This section describes the design and implementation of CoSV, explains how it uses different containers to support diverse model-checking applications, and summarizes its advantages and limitations.

6.3.1 CoSV Design and Implementation

CoSV, an extension and improvement of the original SV [84, 86], generates the same set of VTs, but differs in terms of deployment, scalability, and resource management. CoSV, like SV, exploits the inherent independence of VTs by allowing them to execute in parallel without any inter-VT communication overhead. Therefore, packaging VTs in containers does not impact their behavior, while the isolated environment ensures that they run consistently and securely. The independence of VTs and minimal communication between them make them an excellent fit for orchestrators like Kubernetes, which efficiently manage resources, monitor status, and dynamically deploy or remove VTs as needed.

Figure 6.1 illustrates the architecture of CoSV and the deployment of VTs. In CoSV, there are two types of nodes: Control and Worker. The Control Node acts as the "brain" of the cluster and consists of two main components: the CoSV Coordinator, which configures, generates, and packages all VTs with containers,

and the Kubernetes Control Plane, which schedules and distributes the packaged VTs across all Worker Nodes. CoSV begins by specifying configurations through a static CoSV configuration file that includes all native SV configurations (number of VTs, maximum search depth, etc.) and CoSV-specific configurations (*e.g.*, resources, permissions, working directories, and the external libraries and tools required for each VT). The VT Generator produces a set of VTs whose size is chosen based on the availability of computing resources.

The CoSV Image Builder then packages each VT with all necessary libraries and dependencies (*e.g.*, block devices and file system utilities for Metis) and creates a container specification file, such as a Dockerfile, with instructions on how to automate the VT's environment setup and image building. We publish the resulting container images for each VT to the GitHub Container Registry [63], storing them for easy deployment by Kubernetes, thereby removing the reliance on `scp` and `ssh` for transferring and executing VTs on remote machines.

The CoSV Coordinator then defines batch-style Kubernetes jobs containing everything needed to execute each VT and sends them to the Kubernetes Control Plane for pod creation and scheduling. The Resource Allocator judiciously allocates computing resources (*e.g.*, CPU, memory, disks) to each VT, continuously monitoring available resources (*e.g.*, freed resources from a terminated VT, new nodes, etc.) and dynamically allocating new VTs to maximize scalability and resource efficiency. The Kubernetes Control Plane includes native components such as the API Server, Controller Manager, and Scheduler, which respectively handle communication and requests, store configuration data and state, monitor the cluster's status, and place VTs on available nodes as pods, the smallest deployable unit in Kubernetes. Thanks to the CoSV Control Node, CoSV can build and distribute VTs across multiple Worker Nodes and effectively manage the entire life cycle of these VTs.

Worker Nodes execute VTs packaged in containers and deployed as pods. Each Worker Node can host one or more pods, depending on the available computing resources. Worker Nodes are physical or virtual machines that can be hosted on-premises (*e.g.*, in private data centers), in a research cloud testbed (*e.g.*, Chameleon Cloud [105]), in the public cloud, or in a hybrid-cloud environment. Each pod contains a single container, with each container encapsulating one VT. This provides stronger isolation among VTs and finer control over pod management. In traditional SV, adding a new node to handle more VTs is cumbersome as it requires editing the Swarm configuration file, regenerating the VTs, and using `ssh` and `scp` for file transfer and remote execution, all without visibility into the execution status of the VTs. CoSV, however, can easily add a new Worker Node by running the

existing Kubernetes utility on the Control Node and then deploying new VTs using pre-packaged containers.

The Control Node can monitor all Worker Nodes to identify those that have terminated (*e.g.*, due to assertion failures) and add new VTs accordingly. Some model-checking tasks, like Metis, require a large amount of disk space to store logs and generated files. CoSV uses Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) to manage storage for such VTs. We provision PVs on worker nodes to abstract underlying storage resources, while PVCs enable VTs to request storage configurations, including size and access modes, according to their specific requirements. This decoupling allows CoSV to dynamically provision storage without manual intervention, providing flexibility in handling the large and persistent data produced by VTs.

We have implemented CoSV for two SPIN-based models: Metis and Dining Philosophers. Both models adhere to the architecture and procedure outlined above; the difference lies in specifying their dependencies and resource requirements. Since Metis performs file system checks at the implementation level using extensive C/C++ code, it requires diverse dependencies. CoSV packages all Metis dependencies into a single container, including development libraries (*e.g.*, zlib), block devices (*e.g.*, ramdisks), root permissions, and sufficient memory and storage for state tracking and logging. In contrast, the Dining Philosophers model, implemented solely in Promela, requires few dependencies, generates minimal logs, and maintains a small state footprint. For resource allocation, we ensure that the number of VTs does not exceed the number of CPU cores on each node, so that each VT has at least one core. We allocate memory based on the system state size of the model, where a system state refers to a complete snapshot of all variable values, process control information, and metadata at a given execution point. Each VT is allocated sufficient memory to accommodate the product of the system state size and the maximum search depth, along with additional space for SPIN's hash table, OS usage, and other overhead. Likewise, the disk space allocated to each VT is chosen to be adequate, but not excessive, enabling fine-grained resource management and preventing out-of-space errors.

6.3.2 CoSV Integration with Various Container Types

Given the variety of container types, it is important for CoSV to select the most suitable one based on the model being verified. We implemented CoSV with two representative container backends: Docker and Kata Containers. Both variants, referred to as CoSV-Docker and CoSV-Kata, respectively, use the CoSV design

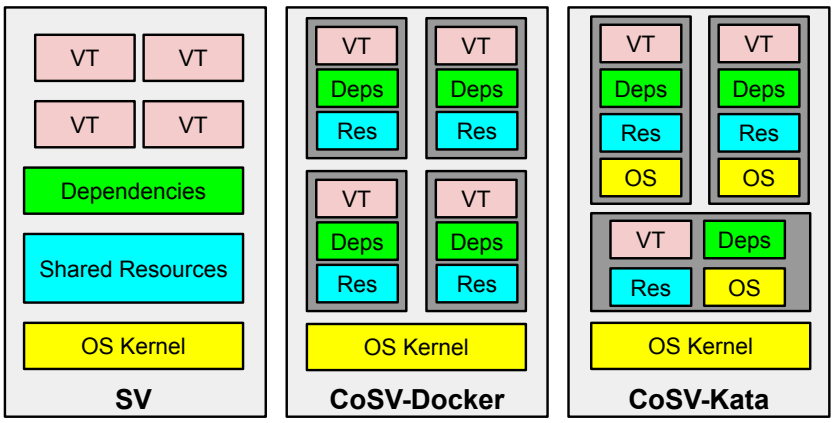


Figure 6.2: VT isolation comparison of SV, CoSV-Docker, and CoSV-Kata in terms of dependencies (Deps), resources (Res), and OS kernel (OS).

described in Section 6.3.1, but employ different container types to provide varying levels of isolation. Docker provides process isolation using Linux namespaces and resource control via cgroups [18], while Kata Containers offer stronger isolation by encapsulating each container within a lightweight VM.

Figure 6.2 illustrates how SV, CoSV-Docker, and CoSV-Kata differ in managing VTs, dependencies, resources, and the OS kernel. In SV, all VTs run in a shared environment, using the same computing resources and host kernel, with dependencies satisfied by the host system. This architecture lacks fault and resource isolation among VTs and requires repeated effort to configure dependencies, such as libraries and tools, on each machine in the cluster. In CoSV-Docker, each VT is packaged with its required dependencies and resources into a container, helping to isolate resource-related issues such as memory errors and contention. This isolation is lightweight, as containers share the host OS kernel, avoiding the overhead of loading and managing a separate kernel for each container. This approach also means, however, that VTs are not isolated from OS kernel-level errors, including kernel panics and crashes: if the host kernel fails, all containers and VTs running on the host will be terminated. To address this issue, CoSV-Kata encapsulates all dependencies and provides resource isolation through a VM, ensuring that each VT runs with its own OS kernel. This also isolates kernel-level errors that may occur during model checking, particularly in low-level system verification tasks.

With CoSV-Docker and CoSV-Kata, CoSV allows one to choose the container type based on the model-checking task at hand, balancing overhead and isolation

needs. In general, CoSV-Docker is suitable for model-checking tasks that operate primarily in user space, while CoSV-Kata is better suited for low-level system verification that may trigger kernel-space restrictions or bugs.

6.3.3 CoSV Advantages and Limitations

To elucidate CoSV's advantages, we compare it with SV across seven dimensions: scalability, fault isolation, runtime environment stability, resource management, deployment, model switchover, and orchestration.

Scalability. SV generates and distributes VTs under the assumption of a static environment, where computing resources remain constant over time. Each VT is allocated fixed resources, and the total number of VTs is predefined. This approach is incompatible with modern computing infrastructures, where resources are constantly changing [9]. In comparison, CoSV leverages containerized VTs that can be easily deployed or removed in response to resource changes. By integrating with an orchestrator to monitor resource availability, CoSV achieves significantly better scalability than SV, particularly in dynamic environments.

Fault Isolation. SV does not account for fault isolation among VTs and assumes they do not interfere with one another. In practice, this is often not the case [154]. Since the verifiers in VTs are implemented in C, they are prone to memory errors such as leaks, especially when the model is improperly specified or contains design flaws. Moreover, since a swarm of VTs aims to collectively explore most of the system state space, some VTs are likely to uncover bugs that trigger system-level errors, which may affect other processes on the same host, as observed in file-system model checking [127, 206, 207]. CoSV provides fault isolation for both user-level errors (*e.g.*, memory leaks) and kernel-level errors (*e.g.*, kernel crashes), using CoSV-Docker and CoSV-Kata, respectively.

Runtime Environment Stability Model checking depends on a stable runtime environment to ensure the reproducibility of detected bugs and counterexamples. SV does not guarantee runtime stability because it relies on a shared, unmanaged execution environment. In such cases, a bug found by a VT might not be reproducible because the runtime environment is not consistent or fixed across executions [127]. CoSV addresses this problem by packaging each VT within a container. This preserves execution consistency across an environment change

(*e.g.*, changing a node) by ensuring that the VT runs in the same isolated and reproducible context.

Resource Management Efficient resource utilization is essential for maximizing the number of concurrently running VTs and achieving the highest possible state exploration rate. In SV, each VT's verifier is configured with a memory limit and monitors its own memory usage to determine if it exceeds that limit. This method alone, however, does not ensure optimal resource utilization. While it prevents VTs from exceeding their limits, it does not guarantee that all available resources are fully utilized. CoSV addresses this inefficiency by dynamically allocating and reclaiming resources across the cluster. It adds or removes VTs based on real-time resource availability to ensure more efficient utilization of system capacity.

Deployment and Model Switchover Traditional SV requires manual setup and static configurations, making VT deployment and model switching inefficient due to the repeated effort needed to configure machines and dependencies. CoSV simplifies deployment by packaging each VT into a container along with its dependencies and environment, requiring only a one-time setup regardless of the number of machines or model switchovers.

Orchestration In standard SV, VTs lack orchestration; their status is not centrally monitored, and completion is signaled only by the presence of a `swarm_done` file. CoSV, in contrast, uses an orchestrator to manage VT lifecycles, providing centralized status tracking, fault recovery, and synchronized termination detection. It supports dynamic deployment, updating, and removal of VTs based on resource availability and workload demands.

CoSV is not without limitations when compared to the standard SV. The most significant limitation is the performance overhead introduced by the use of containers and the orchestrator. In terms of container overhead, CoSV incurs higher overhead than SV across multiple dimensions, including system-level abstraction, network latency, and additional kernel bookkeeping and context-switching costs associated with isolation [140]. Stronger isolation also comes with additional overhead. CoSV-Kata incurs higher overhead than CoSV-Docker because it runs a dedicated kernel for each VT, resulting in increased CPU and memory usage. The additional virtualization layer further introduces latency in network and disk I/O operations. The orchestrator in CoSV introduces overhead by requiring an additional dedicated control node and by adding scheduling latency, resource monitoring load, and periodic health checks, all of which may impact

overall system efficiency. A comprehensive performance evaluation is presented in Section 6.4.

6.4 Evaluation

In this section, we consider the amount of human effort required to deploy CoSV vs. standard SV. We also conduct a performance evaluation and comparison of the two technologies on two SPIN-based models: Metis, a platform for file system model checking, and the Dining Philosophers problem.

6.4.1 CoSV Deployment and Scalability

To compare their deployment processes and scalability, we ran CoSV and SV on three machines in a hybrid cloud environment. For CoSV, we launched the control node and one worker node on our private cloud, and two additional worker nodes on the Chameleon Cloud [105], an open cloud platform. For SV, we used three machines, including one from our private cloud and two from Chameleon Cloud, which were the same machines used for CoSV's worker nodes. We deployed six VTs on each worker node for both CoSV and SV, totaling 18 VTs each. We structured the deployment process into three phases: cluster setup, dependency setup, and VT deployment. We also analyzed scalability in terms of the amount of effort required to add VTs and nodes in dynamic environments, such as cloud-based Spot VMs [205], which utilize idle resources reclaimed from other users.

Table 6.1 outlines the steps required for each deployment stage and the corresponding manual deployment time (*i.e.*, human effort), as reported by two users who performed both SV and CoSV deployment. The *cluster setup stage* establishes communication between machines, allowing them to form a cluster so that VTs can be generated on one node and distributed to others. CoSV sets up the cluster by creating a Kubernetes environment, installing Kubernetes and Calico [185] (for networking) on each node, and joining the worker nodes to the control node. SV needs to set up password-less `ssh` between all nodes in the cluster. We found that CoSV takes slightly less deployment time than SV due to its automated setup procedure and avoidance of manual `ssh` configuration. The *dependency setup stage* configures the environment and installs necessary dependencies on each node to enable them to run VTs. Different models require different sets of dependencies, leading to varying setup times. For this analysis, we estimate the manual effort based on the Metis model. CoSV handles dependency setup by constructing a

container image preloaded with all required components, allowing all VTs to run in the desired environment without additional setup on other machines. SV, on the other hand, must set up dependencies individually on each machine running VTs. Therefore, SV requires 60 minutes of human effort—three times as long as CoSV's 20 minutes—for dependency setup, as it needs to configure all three machines; in contrast, CoSV configures dependencies only once. For VT deployment, CoSV and SV generate VTs in a similar manner but deploy them differently: CoSV uses Kubernetes pods, while SV runs them directly on the machine. Although the deployment approaches differ, the required human effort is similar.

VTs can continue to be scaled post-deployment as new computing resources are added. To add VTs in response to increased CPU availability, CoSV launches new pods with the existing container image, whereas SV requires reconfiguring the swarm, manually updating the script, and dispatching new VTs to the selected node. When new nodes are added, CoSV integrates them into the cluster and deploys VTs without reconfiguring dependencies. With SV, however, the user must configure dependencies, set up password-less `ssh`, and manually deploy VTs, as discussed previously. Adding VTs and adding nodes are both repetitive tasks. Given that VTs can run for long durations and often need to be scaled multiple times, CoSV's more efficient handling of both scaling steps leads to substantial (human) cumulative time savings compared to SV.

Overall, CoSV requires less human deployment time than SV, especially in dynamic environments where new VTs and nodes may be added. Thus, CoSV has better scalability and a more streamlined deployment process than SV. We also found that when switching from the Metis model to the Dining Philosophers model, "model switchover" is easier with CoSV than with SV. This is because CoSV allows the use of a *container template*, requiring updates only to dependencies and resource specifications, without repeating the setup on every node.

6.4.2 Fault Isolation Case Study

A major benefit of using CoSV is *fault isolation*: the ability to prevent faults in one VT from affecting other VTs on the same node. Each VT is designed to explore different states and transitions, so some may trigger faults or encounter resource contention that affects other VTs. We used Metis to demonstrate fault isolation in CoSV, which as our results show, is a capability lacking in standard SV.

Since Metis uses SPIN to check file systems at the implementation level, it can trigger bugs that cause memory errors, file loss, or kernel crashes. We studied fault isolation using two common errors encountered by Metis VTs: (1) a memory

leak leading to an Out-of-Memory (OOM) error, and (2) a kernel crash causing a system failure. The experiments were conducted using four VMs, each equipped with 8 CPU cores, 256 GB of RAM, and a 512 GB disk partition serving as secondary storage for all VT data. One VM acted as the CoSV Control Node, responsible for handling and managing VTs, while the remaining three served as Worker Nodes running VTs for both CoSV and SV. We deployed a total of 18 VTs, with six VTs on each Worker Node. We injected faults into a randomly-chosen VT and observed their impact on the other VTs on the same machine. Ideally, we seek fault isolation: a faulty VT should not affect other VTs by slowing them down or even causing them to prematurely terminate. In the latter case, identifying the faulty VT is difficult because all VTs halt, and no log or execution trail is generated to explain the fault.

Table 6.2 compares the fault-isolation behavior of SV, CoSV-Docker, and CoSV-Kata. Specifically, we injected memory-leak and kernel-crash faults into one VT so that we could investigate their impact on the behavior of the other VTs in the swarm. For memory-leak simulation, we employed two methods: an *OOM Trigger Program* and a *Memory Fill Attack*, both of which induce high memory usage in one VT, triggering OOM (out-of-memory), a likely scenario in Metis and other model checkers due to (real) memory leak bugs and state explosion. Both methods have similar effects. The C-based *OOM Trigger Program* aggressively allocates memory until system resources are exhausted, leading to contention with other processes and VTs. The *Memory Fill Attack* continuously writes to a file in `/tmp` (a RAM-backed directory in our configuration) to exhaust memory and swap space, triggering an OOM event.

As shown in Table 6.2, the other VTs in SV and CoSV responded differently when a memory leak occurs in one VT. Due to the OOM condition caused by the memory leak, the average performance (operations per second) of the other VTs in SV decreased by 52.9% and 44.1% for the *OOM Trigger Program* and *Memory Fill Attack*, respectively. Because of Linux's memory protection mechanisms, other VTs are not forcibly terminated, but do suffer from memory starvation and performance degradation, as there is no isolation among VTs.

Our observations reveal that performance degradation among VTs is highly sensitive to memory usage: other VTs slow down once a single VT starts increasing its memory footprint. This highlights the lack of fault isolation in vanilla SV under resource contention, which in turn causes inefficient resource allocation and degrades the overall performance of the model-checking process. CoSV, whether CoSV-Docker or CoSV-Kata, provides resource isolation among VTs on the same host. As a result, a memory leak in one VT does not impact the resources or

performance of the other VTs.

The other type of fault is a *kernel crash*, which occurs when the operating system's core (the kernel) encounters an unrecoverable error, leading to an unexpected system halt or reboot. This type of fault is common in implementation-level model checking of system software, such as file systems [207, 206]. Indeed, Metis has found a number of file-system bugs that lead to kernel crashes, including memory errors and kernel panics [127]. To evaluate the impact of a kernel crash in one VT on the other VTs, we injected a faulty kernel module that triggers a *kernel panic*—a form of kernel crash—into a single VT and observed its effect on the remaining VTs under SV and CoSV. Table 6.2 also documents the system behavior in response to a kernel crash. When using standard SV with Metis, the system halts immediately after a VT triggers the faulty kernel module. This not only interrupts all other VTs on the same host, but also prevents logging and trace generation, making bug reproduction and analysis more difficult.

CoSV-Docker exhibits the same behavior as SV during a kernel crash: all VTs halt even if the crash did not originate in them. This demonstrates that CoSV with Docker (process-isolated containers with no kernel isolation) cannot protect VTs from kernel-level crash faults, as all CoSV-Docker VTs share the same host kernel. In contrast, in CoSV-Kata, the other VTs are unaffected and continue to function normally after a kernel crash is triggered by one VT. This is expected because each VT on the same host operates within its own isolated OS kernel, preventing a crash in one from affecting the others. CoSV-Kata has proven effective in providing fault isolation during model checking, particularly in scenarios where kernel-level errors may be triggered by the checker.

6.4.3 CoSV for the Dining Philosophers Problem

The Dining Philosophers problem is modeled in SPIN as a collection of circularly arranged concurrent processes, one per philosopher, where each philosopher alternates between an arbitrarily long thinking phase and an eating phase in which it attempts to acquire two forks (one to its left and one to its right) to eat. This classic problem illustrates how synchronization can be used to avoid deadlock and starvation. For this model, SPIN systematically explores all philosopher and fork states across various action interleavings. The number of philosophers largely determines the state-space size, as philosopher-fork configurations grow exponentially with the philosopher count. We ran the Dining Philosophers model with different numbers of philosophers on 18 VTs across three compute nodes, using CoSV and SV to compare performance. We used CoSV-Docker because the

Dining Philosophers model operates entirely in user space and does not trigger kernel-level errors. Of the three nodes, one was an on-premise virtual machine while the other two were Chameleon Cloud instances, illustrating a hybrid cloud environment, with each node running six VTs. SPIN's default maximum depth limit for depth-first search (DFS) is 10,000, but this value restricts state exploration for the Dining Philosophers model, especially for a large number of philosophers; instead we set it to 100,000 for all experiments.

Table 6.3 presents experimental results comparing traditional SV and CoSV-Docker applied to the Dining Philosophers problem. The first column lists the number of philosophers and the second shows the total states explored by SV and CoSV-Docker for each philosopher count. We computed the total number of states by summing the unique system states from SPIN's output across all VTs. Both SV and CoSV-Docker explore the same state space, resulting in identical total-state counts. The elapsed time represents the duration required to complete the longest-running VT. A shorter elapsed time indicates better performance. The exploration rate measures the number of states explored per second, where a higher rate signifies better performance.

As shown in Table 6.3, the performance of SV and CoSV-Docker is nearly identical, with SV outperforming CoSV-Docker by 0.2% to 1.4% in state-exploration rate. This outcome is expected due to the additional overhead from VT containerization, but CoSV significantly reduces VT development and deployment effort, especially in hybrid cloud environments.

6.4.4 CoSV for Metis

The hardware setup for evaluating standard SV, CoSV-Docker, and CoSV-Kata on Metis used the same settings as in the previous fault isolation experiment in Section 6.4.2: three Worker Nodes, each running six VTs (18 VTs in total) for both CoSV and SV, and one additional CoSV-Docker and CoSV-Kata Control Node. We allocated the 18 VTs evenly across the Worker Nodes, with six VTs on each. We evaluated all three approaches by running Metis in swarm-verification mode on ext4 [144], the default file system for many Linux distributions, backed by a RAM disk to ensure high performance. The file system type does not impact the evaluation results since Metis checks properties common to all Linux file systems and explores the same state space regardless of the underlying file system.

Metis explores the state space by performing (mutating) file system operations (*i.e.*, system calls) using SPIN's nondeterministic-loop construct. After each operation, Metis checks whether the abstract state of the file system has been

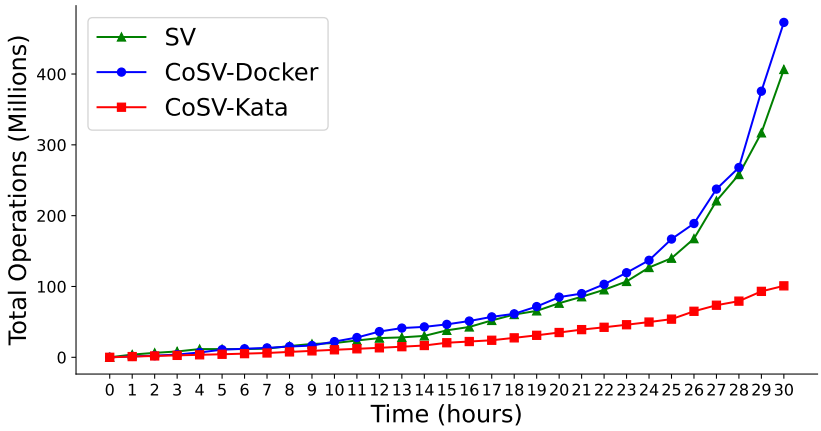


Figure 6.3: Total operations performed (averaged over 3 runs) for the Metis case study: SV vs. CoSV-Docker vs. CoSV-Kata.

visited. The abstract state includes file data, directory structure, and important metadata, but excludes irrelevant fields such as timestamps [127]. Based on this state representation, Metis avoids revisiting already-explored states and instead tries to explore new states in the state space. Our primary performance metrics for comparing SV, CoSV-Docker, and CoSV-Kata are the total number of file-system operations executed and the total number of unique states explored across all VTs. Certain operations result in duplicate abstract states: for example, attempting to create an existing file does not generate a new state. Thus, the number of unique abstract states is always less than or equal to the total number of operations. These two metrics are, however, closely related: a higher number of operations generally leads to a higher number of unique abstract states.

We conducted three runs for each of the three approaches to reduce variance from individual executions, ensuring that all experiments used the same configurations, including the random seed. For each approach, we report the average total number of operations executed and the number of unique abstract states explored across all VTs over the three runs, measured over a 30-hour period. Figure 6.3 illustrates the trend in the total number of file system operations executed. The X-axis represents elapsed time (in hours), and the Y-axis shows the total number of operations (in millions). CoSV-Kata consistently lagged behind both SV and CoSV-Docker, with the divergence becoming noticeable around the 15-hour mark. CoSV-Docker initially maintained an operation rate similar to SV but gradually began to surpass it starting around hour 10. By the end of the 30-hour

period, CoSV-Docker had issued a total of 472.9 million operations, compared to 406.4 million for SV and 100.9 million for CoSV-Kata, amounting to 16.4% more operations than SV and 368.8% more than CoSV-Kata.

We observed similar trends in the number of unique abstract states; in fact, a plot of the unique states is visually identical to Figure 6.3 except for scale, and is thus omitted here due to space constraints. As before, with this metric, CoSV-Docker began to outpace SV from the 10-hour mark onward. By the end of the 30 hours, CoSV-Docker had explored 92.9 million unique states, while SV reached 80.9 million and CoSV-Kata only 21.6 million, reflecting 14.8% higher state coverage than SV and over 330.1% more than CoSV-Kata.

From our performance experiments, it is evident that CoSV-Kata consistently trails behind SV and CoSV-Docker in terms of the total numbers of file system operations and unique abstract states explored. This performance gap primarily stems from the stronger isolation guarantees provided by CoSV-Kata, which incurs higher I/O overheads due to its virtualization of hardware. To quantify this impact, we computed the average file-system operations rate across all three experiments and all 18 VTs over a 30-hour period. CoSV-Kata achieved an average rate of 118 operations/s, approximately 81.4% lower than that of SV (636 operations/s). Despite this lower throughput, the standard deviations were relatively close, 9 operations/s for CoSV-Kata (7.62% of its average rate) versus 15 operations/s for SV (2.35% of its average rate), indicating consistent performance albeit at a reduced rate for CoSV-Kata.

In contrast, CoSV-Docker reached an average rate of 508 operations/s—about 20.1% lower than SV—yet exhibited higher variability (a standard deviation of 43 operations/s, *i.e.*, 8.46% of its average rate) and a maximum observed rate of 677 operations/s, surpassing both SV (664 operations/s) and CoSV-Kata (149 operations/s). This variability, combined with the nondeterministic nature of SPIN's depth-first state space exploration, explains why CoSV-Docker outperformed SV in two out of three experiments, ultimately leading in the aggregate results for both total operations and unique states explored. We conclude that for workloads like Metis, which involve continuous access to physical devices (such as a RAM disk) on the host OS, CoSV-Docker can match or even exceed SV's performance on average, while still offering enhanced container-level isolation.

Importantly, CoSV-Kata represents a meaningful point on the performance-isolation spectrum. While it may not deliver the same raw throughput as SV or CoSV-Docker for I/O-intensive workloads, it excels in providing robust isolation through hardware-assisted virtualization. This makes CoSV-Kata a compelling choice for security-critical or multi-tenant environments, particularly in low-level

system verification models where strong isolation is essential. These results underscore the flexibility of the CoSV framework in accommodating diverse deployment goals, whether maximizing performance, strengthening isolation, or achieving a balanced trade-off between the two.

6.5 Chapter Conclusion

We have presented *Containerized Swarm Verification* (CoSV), an extension and modernization of SV in which verification tasks (VTs) are executed in isolated containers managed by an orchestrator. CoSV addresses various limitations of SV by improving deployment, scalability, fault isolation, and both resource and lifecycle management. We developed two variants of CoSV, CoSV-Docker and CoSV-Kata, leveraging different container technologies to offer varying levels of isolation for diverse model-checking applications. Our experimental results demonstrate that: (i) CoSV facilitates the deployment of VTs across multi-core servers and hybrid cloud environments, (ii) CoSV achieves performance comparable to standard SV while incurring only limited overhead, and, importantly, (iii) CoSV provides fault isolation among VTs, preventing faults at both the user and kernel levels from propagating throughout the swarm. We plan to open-source all CoSV artifacts, including CoSV-Docker and CoSV-Kata, with the goal of advancing scalable and parallel model checking in both academic and industrial settings.

Deployment Stage	**SV Step**	**SV Time (min)**	**CoSV Step**	**CoSV Time (min)**
Cluster Setup	Set up password-less `ssh` between all nodes	20	Build Kubernetes cluster and join worker nodes to the control node	15
Dependency Setup	Install dependencies on every machine running VTs	60	One-time effort to build container image	20
VT Deployment	Run VTs directly on nodes	10	Launch pods using the pre-built image	10
Adding VTs (repetitive)	Reconfigure and manually edit Swarm script	15	Create pods from existing container images	3
Adding Nodes (repetitive)	Set up dependencies and `ssh` connection for every new node	25	Join new nodes to cluster without the need to set up dependencies	5

Table 6.1: Comparison of SV and CoSV deployment steps with estimated human deployment times. Tasks such as adding nodes and VTs are repetitive, with time proportional to the number of components. For example, adding nodes takes approximately $25 \times N$ minutes, and adding VTs takes $15 \times V$ minutes, where N is the number of nodes and V is the number of VTs. Note: SSH setup is included in both the initial cluster setup and the node addition stage, and is required for each node.

Fault Type	**Fault Injection**	**SV**	**CoSV-Docker**	**CoSV-Kata**
Memory Leak	Out-Of-Memory Trigger Program	Performance degraded by 52.9%	**No performance impact**	**No performance impact**
	Memory Fill Attack	Performance degraded by 44.1%	**No performance impact**	**No performance impact**
Kernel Crash	Kernel-Crashing Module	Stopped immediately	Stopped immediately	**Operated normally**

Table 6.2: Behavioral comparison of other VTs after injecting a fault into one VT under SV, CoSV-Docker, and CoSV-Kata. **Bold** entries indicate fault-isolated results.

No. of Philos.	**Total States**	**SV**		**CoSV-Docker**	
		Elapsed Time (secs)	**Exploration Rate (states/sec)**	**Elapsed Time (secs)**	**Exploration Rate (states/sec)**
10	521,156,491	279	1,867,944	283	1,841,542
20	1,554,682,594	1,570	990,244	1,573	988,355
30	1,651,535,265	2,652	622,751	2,657	621,579

Table 6.3: Performance comparison of SV and CoSV for varying nos. of philosophers.

Chapter 7

Conclusions

Our thesis is that ensuring file system reliability through testing requires a multi-faceted approach: effective coverage metrics to assess and enhance existing tests, a new model checking framework for comprehensive and versatile testing, and scalable techniques for parallel testing. As new file systems are developed and bugs continue to emerge, testing must be an ongoing effort that evolves to accommodate new systems and features. File system testing faces three key problems that require sustained effort and effective solutions: defining measurable coverage metrics, enabling thorough model checking with minimal manual effort, and developing scalable approaches for deploying tests across computing resources.

To address the problem of coverage metrics, we have defined input and output coverage and implemented IOCov framework to compute them. We demonstrate how these metrics can identify untested parts of file system testing tools and provide insights for improvement. We further introduce CM-IOCov, which enhances input coverage for existing crash consistency testing and leads to improved bug detection.

To address the problem of file system model checking, we developed Metis, which combines implementation-level model checking with differential testing. Metis generates diverse inputs to explore a wide range of file system states and compares their behaviors against a reference file system, RefFS. We demonstrate the performance of Metis and RefFS and how they together can detect bugs across multiple file systems. Metis uses Swarm verification to explore the file system's large state space in parallel, achieving broader coverage within a limited time.

To further improve model checking scalability, we created CoSV to simplify deployment and provide fault isolation in Swarm verification. CoSV containerizes verification tasks and manages them with an orchestrator, allowing each task to be configured once and easily scaled out across multiple machines. CoSV isolates

verification tasks at different levels: resource level isolation and kernel level isolation, suited for user space and kernel space model checking, respectively.

In sum, through three dimensions of research advancement—*coverage*, *framework*, and *scalability*—we improve the effectiveness and practicality of file system testing and model checking, enabling the detection of more bugs and ultimately enhancing file system and overall system reliability.

7.1 Future Work

This section outlines future work by providing an overview of open research problems that can be derived from this thesis. These future directions aim to enhance the capabilities of model checking and testing from multiple perspectives, including universality, usability, effectiveness, and scalability, to facilitate bug detection and further improve the reliability of file systems and broader systems.

7.1.1 File System Model Checking and Bug Detection

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 [173]. 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 [165, 193, 148]. 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 [74, 134], 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 [34, 176]. 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 [151, 115]. For detecting concurrency bugs, file system operations should be executed concurrently using multiple threads. Additionally, checking file system concurrency requires incorporating thread interleaving states [66, 201] 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 [170]. Further exploration of root-cause analysis in conjunction with model checking is a worthwhile avenue of research.

7.1.2 Model Checking Applications and Enhancements

Model checking beyond file systems. Model checking has been applied to verify numerous types of software, and its potential is not limited to file systems. Our approach of integrating implementation-level model checking with differen-

tial testing shows potential for application to other software systems as well, as long as the systems under verification meet the following criteria: (1) their behaviors can be compared across different implementations or versions; (2) their internal states can be represented and explored by feeding inputs; and (3) their correctness can be observed through those states. These software systems include databases [97], compilers [120], and network protocols implementations (*e.g.*, TLS libraries [137]). Moreover, this model checking approach can also be applied to verify algorithm implementations, including concurrency and synchronization algorithms [45]. It can also help detect and prevent security issues such as network intrusions [37, 36] and backdoor attacks [136, 135], thereby improving the overall reliability and security of computing systems.

We hope our approach opens the door to broader adoption of model checking for previously unexplored tasks, enabling the discovery of bugs and vulnerabilities across a wider range of applications, protocols, and both software and hardware platforms.

Usability and scalability improvements in model checking. Applying model checking requires constructing a model representation and capturing state information, which can be difficult to obtain and thus limits its usability. A promising future direction is to automatically extract model and state representations from system code [88] or auxiliary sources such as documentation and logs, potentially with the help of large language models (LLMs) [75]. Our experience with file system abstract state indicates that retaining only essential properties and omitting potentially noisy ones is key to reducing the state space and improving bug detection accuracy [182, 127]. Therefore, automatically identifying essential properties and critical states worth exploring is a valuable research direction for improving the usability of model checking. Additionally, the large size of state representations and the state explosion problem introduce significant overhead, potentially leading to memory and storage issues, slower state exploration, and missed critical property checks [182, 127, 123, 124]. Future research on data compression algorithms [142, 141, 143] to reduce state size, and optimization techniques such as advanced partial order reduction [55] to shrink the state space, represents a promising direction for addressing these challenges.

Enhancing the scalability of model checking is an ongoing effort that involves not only improving existing techniques such as Swarm verification, but also designing new scalable approaches that address its limitations. For example, coordinated verification tasks can reduce redundant state exploration through inter-task com-

munication. Furthermore, leveraging advanced hardware algorithms (*e.g.*, CPU scheduling [215, 216], NUMA-aware optimizations [49], and network optimizations [35]) and platforms (*e.g.*, GPUs [42] and FPGAs [38, 159]) can further increase parallelism.

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-ext*2.

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

[6] 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.

[7] Mário Antunes, Tyler Estro, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Kneeliv-

erse: A universal knee-detection library for performance curves. *SoftwareX*, 30:102161, 2025.

[8] 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.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

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

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

[12] 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.

[13] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 201–207, Pune, India, October 2017. Springer.

[14] Jiri Barnat, Vincent Bloemen, Alexandre Duret-Lutz, Alfons Laarman, Laure Petrucci, Jaco van de Pol, and Etienne Renault. Parallel model checking algorithms for linear-time temporal logic. *Handbook of Parallel Constraint Reasoning*, pages 457–507, 2018.

[15] Jiri Barnat, Lubos Brim, and Jakub Chaloupka. Parallel breadth-first search LTL model-checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 106–115, Montreal, Canada, October 2003. IEEE.

[16] Jiří Barnat, Luboš Brim, and Petr Ročkai. Scalable multi-core LTL model-checking. In *Model Checking Software: Proceedings the 14th International SPIN Workshop*, pages 187–203, Berlin, Germany, July 2007. Springer.

[17] Jiri Barnat, Lubos Brim, and Jitka Stříbrná. Distributed LTL model-checking in SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN)*, pages 200–216, Toronto, Canada, May 2001. Springer.

[18] David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

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

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

[21] 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*.

[22] 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.

[23] 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.

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

[25] 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.

[26] Ethan Burns and Rong Zhou. Parallel model checking using abstraction. In *Proceedings of the 19th International SPIN Workshop on Model Checking Software (SPIN)*, pages 172–190, Oxford, UK, July 2012. Springer.

[27] 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.

[28] 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.

[29] 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*.

[30] 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.

[31] 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.

[32] 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.

[33] 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. ACM.

[34] 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.

[35] Jingdi Chen, Yimeng Wang, and Tian Lan. Bringing fairness to actor-critic reinforcement learning for network utility optimization. In *Proceedings of the 40th IEEE Conference on Computer Communications (INFOCOM)*, pages 1–10, Vancouver, BC, Canada, May 2021. IEEE.

[36] Jingdi Chen, Lei Zhang, Joseph Riem, Gina Adam, Nathaniel D. Bastian, and Tian Lan. Explainable learning-based intrusion detection supported by memristors. In *IEEE Conference on Artificial Intelligence (CAI)*, pages 195–196, Santa Clara, CA, USA, June 2023. IEEE.

[37] Jingdi Chen, Lei Zhang, Joseph Riem, Gina Adam, Nathaniel D. Bastian, and Tian Lan. RIDE: Real-time intrusion detection via explainable machine learning implemented in a memristor hardware architecture. In *IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8, Tampa, FL, USA, November 2023. IEEE.

[38] Shenghsun Cho, Michael Ferdman, and Peter A. Milder. FPGASwarm: High throughput model checking on FPGAs. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 435–442, Dublin, Ireland, August 2018. IEEE Computer Society.

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

[40] 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.

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

[42] Richard DeFrancisco, Shenghsun Cho, Michael Ferdman, and Scott A Smolka. Swarm model checking on the GPU. *International Journal on Software Tools for Technology Transfer*, 22:583–599, 2020.

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

[44] 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.

[45] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

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

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

[48] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[49] Andi Drebes, Antoniu Pop, Karine Heydemann, Nathalie Drach, and Albert Cohen. Numa-aware scheduling and memory allocation for data-flow task-parallel applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–2, Barcelona, Spain, March 2016. ACM.

[50] 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.

[51] Mike Eisler. NFS version 4. *https://www.usenix.org/legacy/event/lisa*05*/htg/eisler.pdf*.

[52] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Guiding simulations of multi-tier storage caches using knee detection. In *31st Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '23)*, Stony Brook, NY, October 2023. IEEE Computer Society.

[53] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Accelerating multi-tier storage cache simulations using knee detection. *Performance Evaluation*, 164:102410, May 2024.

[54] Ioannis Filippidis and Gerard J. Holzmann. An improvement of the piggyback algorithm for parallel model checking. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software (SPIN)*, pages 48–57, San Jose, CA, USA, July 2014. Springer.

[55] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 110–121, Long Beach, CA, January 2005. ACM.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 8th International SPIN workshop on Model Checking Software (SPIN)*, pages 217–234, Toronto, Canada, May 2001. Springer.

[62] 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.

[63] Github container registry, 2025. *https://github.blog/news-insights/product-news/introducing-github-container-registry/*.

[64] 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.

[65] 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.

[66] 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.

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

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

[69] 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.

[70] 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.

[71] 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. ACM.

[72] 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.

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

[74] 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.

[75] Mohammad Saqib Hasan, Sayontan Ghosh, Dhruv Verma, Geoff Kuenning, Erez Zadok, Scott Smolka, and Niranjan Balasubramanian. Handling open-vocabulary constructs in formalizing specifications: Retrieval augmented parsing with expert knowledge. In *Proceedings of the First Conference on Language Modeling (COLM 2024)*, Philadelphia, PA, October 2024.

[76] 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.

[77] 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.

[78] 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*.

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

[80] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 131–147, Berlin, Heidelberg, 2000. Springer-Verlag.

[81] Gerard J. Holzmann. Parallelizing the spin model checker. In *Proceedings of the 19th International SPIN Workshop on Model Checking Software (SPIN)*, pages 155–171, Oxford, UK, July 2012. Springer.

[82] Gerard J. Holzmann. Swarm verification GitHub repository, 2023. *https://github.com/nimble-code/Swarm*.

[83] 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.

[84] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–6. IEEE, 2008.

[85] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In *Proceedings of the 15th International SPIN workshop on Model Checking Software (SPIN)*, pages 134–143, Los Angeles, CA, USA, August 2008. Springer.

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

[87] Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis. On nested depth first search. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 32, pages 23–31. American Mathematical Society, 1997.

[88] Muhammad Iqbal Hossain and Nahida Sultana Chowdhury. A practical approach on model checking with modex and spin. *International Journal of Electrical and Computer Sciences*, 11:1–7, 2011.

[89] 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.

[90] 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.

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

[92] 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.

[93] Omar Inverso and Catia Trubiani. Parallel and distributed bounded model checking of multi-threaded programs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 202–216, San Diego, California, USA, February 2020. ACM.

[94] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations*

of Software Engineering (ESEC/FSE), pages 955–963, Tallinn, Estonia, August 2019. ACM.

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

[96] 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.

[97] Zu-Ming Jiang and Zhendong Su. Detecting logic bugs in database engines via equivalent expression transformation. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 821–835, Santa Clara, CA, USA, July 2024. USENIX.

[98] 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.

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

[100] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[101] 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.

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

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

[104] Kata containers, 2025. *https://katacontainers.io/*.

[105] 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.

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

[107] 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.

[108] 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.

[109] 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.

[110] Teemu Koponen, Pasi Eronen, and Mikko Särelä. Resilient connections for SSH and TLS. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 329–340, Boston, MA, USA, May 2006. USENIX.

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

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

[113] Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PMDC)*, pages 19–34, London, UK, September 2004. Elsevier.

[114] 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.

[115] 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. ACM.

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

[117] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[118] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *International SPIN Workshop on Model Checking of Software*, pages 22–39, Trento, Italy, July 1999. Springer.

[119] Baokun Li, Ritesh Harjani, and Theodore Ts'o. Ext4: add inode table check in __ext4_get_inode_loc to avoid possible infinite loop, 2022. *https://github.com/torvalds/linux/commit/eee*22187*b*53611*e*173161*e*38*f*61*de*1*c*7*ecbeb*876.

[120] Shaohua Li and Zhendong Su. Finding unstable code via compiler-driven differential testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 238–251, Vancouver, BC, Canada, March 2023. ACM.

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

[122] Jianxiao Liu, Zonglin Tian, Yifei Liu, and Liang Zhao. Research of web service recommendation using bayesian network reasoning. In *Proceedings of the 15th International Conference on Services Computing (SCC '18)*, volume 10969, pages 19–35, Seattle, WA, June 2018. Springer.

[123] Yifei Liu. Model-checking support for file system development. Technical Report FSL-22-01, Computer Science Department, Stony Brook University, January 2022.

[124] Yifei Liu. Towards efficient, scalable, and versatile file system model checking. Technical Report FSL-24-04, Computer Science Department, Stony Brook University, November 2024.

[125] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott Smolka, Wei Su, and Erez Zadok. Artifact package: the Metis file system model checking framework, January 2024.

[126] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott Smolka, Wei Su, and Erez Zadok. Artifact package: the RefFS reference file system for the Metis model checking framework, January 2024.

[127] 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.

[128] 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.

[129] Yifei Liu, Geoff Kuenning, Md. Kamal Parvez, Scott Smolka, and Erez Zadok. Enhanced file system testing through input and output coverage. In *Proceedings of the 18th ACM International Systems and Storage Conference (SYSTOR '25)*, Virtual Event, September 2025. ACM. To Appear.

[130] 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.

[131] 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.

[132] Yu Liu, Yangtao Wang, Ke Zhou, Yujuan Yang, Yifei Liu, Jingkuan Song, and Zhili Xiao. A framework for image dark data assessment. In *Proceedings of the 3rd APWeb-WAIM joint conference on Web and Big Data (APWeb-WAIM '19)*, volume 11641, pages 3–18, Chengdu, China, August 2019. Springer. Won Best Paper Runner-Up.

[133] 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.

[134] 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.

[135] Weimin Lyu, Lu Pang, Tengfei Ma, Haibin Ling, and Chao Chen. TrojVLM: Backdoor attack against vision language models. In *European Conference on Computer Vision (ECCV)*, pages 467–483, Milan, Italy, September 2024. Springer.

[136] Weimin Lyu, Jiachen Yao, Saumya Gupta, Lu Pang, Tao Sun, Lingjie Yi, Lijie Hu, Haibin Ling, and Chao Chen. Backdooring vision-language models with out-of-distribution data. *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, April 2025.

[137] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. TLS-Anvil: Adapting combinatorial testing for TLS libraries. In *Proceedings of the 31st USENIX Security Symposium*, pages 215–232, Boston, MA, August 2022. USENIX Association.

[138] Yashwant K Malaiya, Naixin Li, Jim Bieman, Rick Karcich, and Bob Skibbe. The relationship between test coverage and reliability. In *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE)*, pages 186–195, Monterey, CA, USA, November 1994. IEEE Computer Society.

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

[140] 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.

[141] Yu Mao, Yufei Cui, Tei-Wei Kuo, and Chun Jason Xue. Accelerating general-purpose lossless compression via simple and scalable parameterization. In *Proceedings of the 30th ACM International Conference on Multimedia (MM)*, pages 3205–3213, Lisboa, Portugal, October 2022. ACM.

[142] Yu Mao, Yufei Cui, Tei-Wei Kuo, and Chun Jason Xue. TRACE: A fast transformer-based general-purpose lossless compressor. In *Proceedings of the ACM Web Conference (WWW)*, pages 1829–1838, Virtual Event, Lyon, France, April 2022. ACM.

[143] Yu Mao, Jingzong Li, Yufei Cui, and Jason Chun Xue. Faster and stronger lossless compression with optimized autoregressive framework. In *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, July 2023. IEEE.

[144] 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.

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

[146] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M Kirby, and Ganesh Gopalakrishnan. Parallel and distributed model checking in Eddy. *International Journal on Software Tools for Technology Transfer*, 11:13–25, 2009.

[147] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. CNS-Bench: 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.

[148] 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.

[149] 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.

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

[151] 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.

[152] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Crashmonkey and ACE: Systematically testing file-system crash consistency. *ACM Transactions on Storage (TOS)*, 15(2):1–34, 2019.

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

[154] 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.

[155] 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.

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

[157] 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.

[158] 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.

[159] Mrunal Patel, Shenghsun Cho, Michael Ferdman, and Peter Milder. Runtime-programmable pipelines for model checkers on FPGAs. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 51–58, Barcelona, Spain, September 2019. IEEE Computer Society.

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

[161] 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.

[162] 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.

[163] 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.

[164] Ajitha Rajan. Coverage metrics to measure adequacy of black-box test suites. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 335–338, Tokyo, Japan, September 2006. IEEE Computer Society.

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

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

[167] 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.

[168] 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.

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

[170] 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.

[171] 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.

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

[173] 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.

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

[175] Pujuan Shi, Yihang Fang, Chengda Lin, Yifei Liu, and Ruifang Zhai. A new line detection algorithm - automatic measurement of character parameter of rapeseed plant by lsd. In *Proceedings of the Fourth International Conference on Agro-Geoinformatics (Agro-geoinformatics '15)*, pages 257–262, Istanbul, Turkey, July 2015. IEEE.

[176] 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.

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

[178] Hemanthkumar Sivaraj and Ganesh Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking (PDMC)*, pages 51–67, Boulder, Colorado, USA, July 2003. Elsevier.

[179] 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.

[180] Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. *Formal Methods in System Design*, 18:117–129, 2001.

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

[182] 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.

[183] Alexander Thomson and Daniel J Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015. USENIX Association.

[184] 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.

[185] Tigera. Project calico, 2025. *https://www.projectcalico.org*.

[186] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. BugZoo: A platform for studying software bugs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE)*, pages 446–447, Gothenburg, Sweden, May 2018. ACM.

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

[188] 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.

[189] 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.

[190] 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/*.

[191] 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.

[192] Yangtao Wang, Yu Liu, Yifei Liu, Ke Zhou, Yujuan Yang, Jiangfeng Zeng, Xiaodong Xu, and Zhili Xiao. Analysis and management to hash-based graph and rank. In *Proceedings of the 3rd APWeb-WAIM joint conference on Web and Big Data (APWeb-WAIM '19)*, volume 11641, pages 289–296, Chengdu, China, August 2019. Springer.

[193] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 307–320, Seattle, WA, November 2006. ACM SIGOPS.

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

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

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

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

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

[199] 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.

[200] 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.

[201] 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.

[202] 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.

[203] 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.

[204] Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. Test coverage metrics for the network. In *Proceedings of the ACM SIGCOMM Conference*, pages 775–787, Virtual Event, August 2021. ACM.

[205] Fangkai Yang, Bowen Pang, Jue Zhang, Bo Qiao, Lu Wang, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, et al. Spot virtual machine eviction prediction in Microsoft cloud. In *Companion Proceedings of the Web Conference 2022*, pages 152–156, Virtual Event / Lyon, France, April 2022. ACM.

[206] 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.

[207] 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.

[208] 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.

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

[210] 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.

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

[212] 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.

[213] 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.

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

[215] Ti Zhou and Man Lin. Cpu frequency scheduling of real-time applications on embedded devices with temporal encoding-based deep reinforcement learning. *Journal of Systems Architecture*, 142:102955, 2023.

[216] Ti Zhou, Haoyu Wang, Xinyi Li, and Man Lin. Profiling and understanding cpu power management in linux. In *IEEE Smart World Congress (SWC)*, pages 1–8, Portsmouth, UK, August 2023. IEEE.

[217] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

[218] 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.