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

Yifei Liu and Manish Adkar, *Stony Brook University;*
Gerard Holzmann, *Nimble Research;* Geoff Kuenning, *Harvey Mudd College;*
Pei Liu, Scott A. Smolka, Wei Su, and Erez Zadok, *Stony Brook University*

https://www.usenix.org/conference/fast24/presentation/liu-yifei

This paper is included in the Proceedings of the
22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings
of the 22nd USENIX Conference on
File and Storage Technologies
is sponsored by

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

Yifei Liu, Manish Adkar, Gerard Holzmann*, Geoff Kuenning+, Pei Liu,
Scott A. Smolka, Wei Su, and Erez Zadok
*Stony Brook University, *Nimble Research, +Harvey Mudd College*

Abstract

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 12 bugs from five other file systems, five of which were confirmed and with one fixed and integrated into Linux.

1 Introduction

File system testing is an essential technique for finding bugs [43] and enhancing overall system reliability [27], as file-system bugs can have severe consequences [53, 92]. Effective testing of file systems is challenging, however, due to their inherent complexity [4], including many corner cases [87], myriad functionalities [8], and consistency requirements (*e.g.*, crash consistency [64, 72]). Developers have created various testing technologies [59, 71, 86] for file systems, but new bugs (both in-kernel and non-kernel) continue to emerge on a regular basis [42, 43, 85].

To expose a file-system bug, a testing tool must execute a particular system call using specific inputs on a given file-system state [52, 53, 87]. For example, identifying a well-known Ext4 bug [48] 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 [9, 52] 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 [12, 43, 59, 85]. For example, handwritten regression tools like xfstests [71] can achieve good test coverage of specific file-system features [4, 58], but do not comprehensively cover syscall inputs; similarly, fuzzing techniques (*e.g.*, Syzkaller [25]) are designed to maximize code—not input—coverage [40].

Both the input and state spaces of file systems are too vast to be completely explored and tested [10, 21], so it is better to leverage finite resources by focusing on the most pertinent inputs and states [52, 86, 88]. 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 [67]. Existing testing technologies also lack the versatility to test specific inputs and states [25, 59, 71]. Thus, new testing tools and techniques are needed [52, 53] to avoid under-testing (which could miss potential bugs) or over-testing (which wastes resources that may be better deployed elsewhere).

This paper presents Metis, a novel model-checking framework that enables thorough and versatile input and state space exploration of file systems. Metis runs two file systems concurrently: a file system under test and a reference file system to compare against [26]. 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 [31]).

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 [31]. 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 [16, 31] file systems. It checks file-system implementations directly, eliminating the need to build a formal model [61]. To manage large file-system input and state spaces, Metis enables parallel and distributed exploration [33] 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 [43], that require kernel instrumentation and utilities. Nevertheless, the quality of the reference file system is pivotal for assessing the behavior of other file systems [26]. 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 [73, 86]. 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 [25, 59, 71]. 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, 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 12 potential bugs across five file systems. Of these, 10 were confirmed as previously unknown bugs, five of which were confirmed by developers as real bugs. Moreover, one of those bugs—which the developers confirmed existed for 16 years—and the fix we provided, was recently integrated into mainline Linux.

In sum, this paper makes the following contributions:

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

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 [9, 57] 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 [12, 59, 67]. 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 [16, 77]. The model is typically a state machine, and the properties, usually expressed in temporal logic, are checked using state-space exploration [15]; here, each state represents a snapshot of the system under investigation. To automate this process, model checkers (*e.g.*, SPIN [31]) 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 [86, 87]. Thus, recent work on implementation-level model checking [86, 87] seeks to check the implementation directly (without a model). Such approaches [86] 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 [86, 87]. 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 [22, 81]. Thoroughly testing file system inputs is challenging. While file-system–related syscalls represent only a subset of all Linux syscalls [7, 74], each syscall has multiple arguments, and the potential value range for these arguments is vast [52, 74]. 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 [25, 37]. Instead, a sensible approach [45, 52] is to *segment* a large input space into multiple, disjoint input partitions—called *input space partitioning* [39, 52, 78]. How

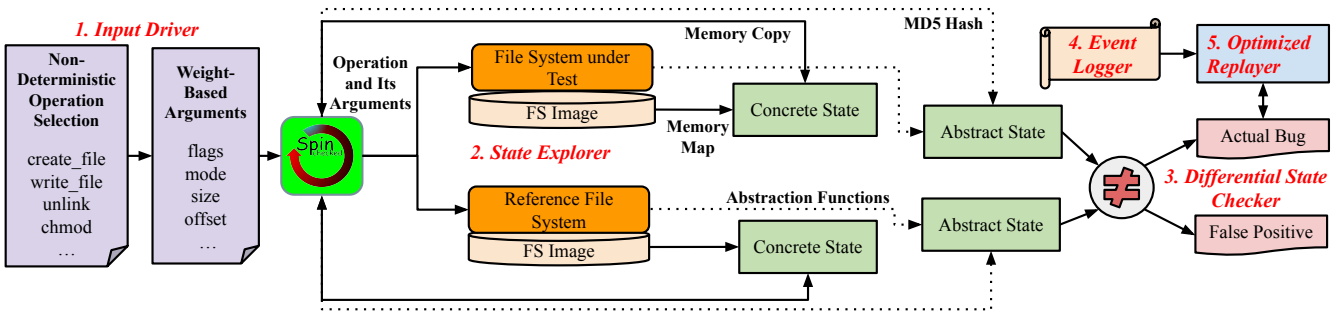

Figure 1: Metis architecture and components. From left to right, Metis generates syscalls and their arguments that are executed by both file systems, determines resulting states, and checks for discrepancies between states. The Logger records all the operations for convenient bug replay by the Replayer. The SPIN model checker stores previous state information for state exploration.

much a testing tool examines input partitions is called *input coverage* [30, 45, 75]. 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 [7, 52, 74]: (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 [38]). 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 [21, 73]. Comprehensive state exploration is important as certain bugs manifest exclusively under specific states [48, 53, 76]. Numerous file system states can be explored when some existing testing approaches [59, 71] 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 [86]. 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 [21]. 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 [16]. Conversely, an overly narrow definition can skip key states and potentially miss defects [11]. 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 [31] 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 [16]. 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 [33].

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 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 (§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 [31] to combine input selection with state exploration. The State Explorer (§3.2) extracts concrete and abstract states from both file systems and interfaces with SPIN to explore new states. The Differential State Checker (§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 (§3.4) help analyze reported discrepancies and reproduce potential bugs more efficiently.

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 [26] (*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 (§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 (§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 [7, 52]. 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 [38], 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 [59], it is used infrequently for POSIX compliance [67]. Due to disk I/O's slow speed, many tests focus on small write sizes [12]. However, testing larger sizes can uncover size-specific bugs [67, 76]. Our objective is to ensure that Metis remains versatile and to allow one to adjust the input weights in line with the test focus.

3.2 State Exploration and Tracking

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 [15]. Metis supports depth-first search (DFS) as the main search algorithm.

The State Explorer relies on the SPIN model checker [31] 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 [11]. 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

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 1: Examples of false positives identified and addressed by Metis.

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 [55]) 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 [2]) 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 [17, 84], VM snapshotting [44, 46], and LightVM [54]. None of these approaches were effective due to their functional deficiencies or inefficient performance. For those reasons, we adopted the approach presented in [73] 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 (§5.2), especially coupled with our specialized RefFS (§4).

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 [12], such as kernel crashes. We thus needed a dedicated checker to identify cases where file systems fail silently [43] (*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 [59, 67]: they are difficult to adapt to continuously-evolving file systems.

File systems vary considerably in terms of their developmental stages [53, 90]: mature file systems are typically more stable than new, emerging, or less popular ones [53]. Yet many still share common (POSIX) features and data-integrity requirements. Therefore, we rely on a *differential testing* approach [56], 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 1 summarizes several such cases including their problems, causes, and solutions.

All these discrepancies arose due to different file system designs and implementations. For instance, Ext4 has a special

`lost+found` directory and computes directory sizes by a multiple of the block size. In contrast, other file systems report sizes by the number of active entries and do not have a `lost+found` directory. Despite the same device sizes for different file systems, the available space varies due to different utilized and reserved space (*e.g.*, for metadata). To address this, we equalize free space among file systems by creating dummy files based on the differences in their available spaces.

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

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 [3]. So we needed a way to reproduce a discrepancy quickly. Existing test-case minimization techniques [43, 91] 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 [43, 59] 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.

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 [31]. 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 [21]. Exploring this space using a single Metis process (called a *verification task*, or VT) requires significant time.

To parallelize the state-space exploration [32] we use Swarm verification [33], 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 [33], 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 [33]. We deployed multiple VTs on several multi-core machines and evaluated Metis extensively under Swarm verification (§5.2).

3.6 Implementation Details

Metis uses SPIN to achieve basic model-checking functions. The Promela modeling language [31] 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.

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 [26, 56], 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 [6], 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 [14]. 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` [28]) 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.

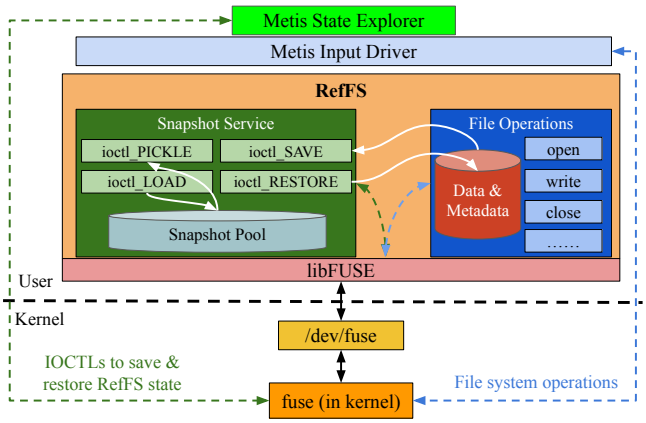

Figure 2: RefFS architecture and its interaction with Metis and kernel space. RefFS supports standard POSIX operations and provides snapshot services with a snapshot pool and four new APIs.

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 [47, 59], Metis cannot find bugs that arise solely during system crashes. We plan to provide the option of invoking utilities such as `fsck` [63] 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 [83]). 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 [69].

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 [55]. 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 [14, 72], 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 [5]. 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 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.

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 [68] and ZFS [8], 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 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 §5.1.) **(2)** What is Metis's performance? How does it scale with the number of VTs when using Swarm verification? (See §5.2.) **(3)** What is RefFS's performance compared to other file systems? How reliable and stable is RefFS, as Metis's reference file system? (See §5.3.) **(4)** With RefFS set as the reference file system, does Metis find bugs in existing Linux file systems? (See §5.4.)

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.

5.1 Test Input Coverage

We assessed input coverage (§2) for Metis and other file system tests on two dimensions: completeness and versatility. Completeness considers whether a testing tool covers all input partitions (§2) in test cases. Versatility is the ability to tailor test cases for any desired input coverage. Metis outperforms existing checkers and a fuzzer [25] on both dimensions.

Comparison with existing testing tools. We selected three testing tools, each representing a unique technique: CrashMonkey [59] for automatic test generation, xfstests [71] for (hand-written) regression testing, and Syzkaller [25] for fuzzing.

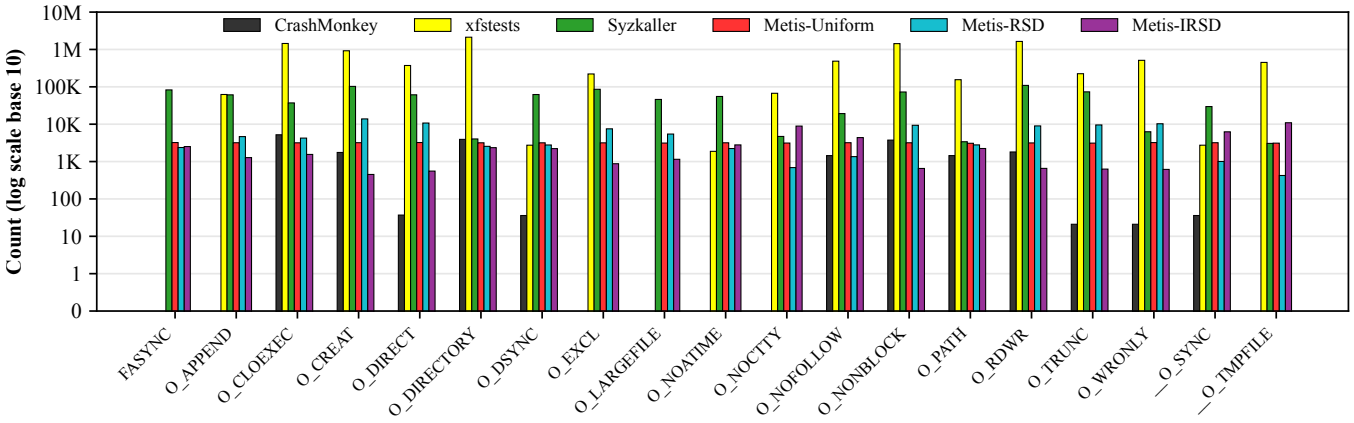

Figure 3: Input coverage counts (log_{10}, y-axis) of `open` *flags (x-axis) for CrashMonkey, xfstests, Syzkaller, and Metis with 3 different weight distributions.*

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 [60].

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 [52] 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 3 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 [66]), and IRSD (Inverse Rank-Size Distribution [62]). 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 3, 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 [79]. 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 3 shows that `O_CREAT` (the most common `open` flag in the kernel source) is indeed tested most often in Metis-RSD and least in Metis-IRSD. `__O_TMPFILE`, the least-frequent flag, exhibits the opposite trend. Other tools lack the versatility to adapt their test input partitions to the desired amount of testing.

Moreover, we observed that xfstests tested certain input values (*e.g.*, `O_DIRECTORY`) millions of times while others (*e.g.*, `FASYNC`) are not tested at all. However, other tools sometimes have a higher total operation count than Metis because Metis has to unmount and remount the file system to achieve state tracking and verify state equality after each operation, slowing its syscall execution speed. Given the essential role of unmount/mount for state tracking (§3.2) and the need for state comparison (§3.3), we use Swarm verification to improve the overall operation efficiency (§3.5).

Input coverage for write size. Figure 4 shows the input coverage for the `write` size (requested byte count). The x-axis represents the log_2 of the size, corresponding to the `write` size partitions (see §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

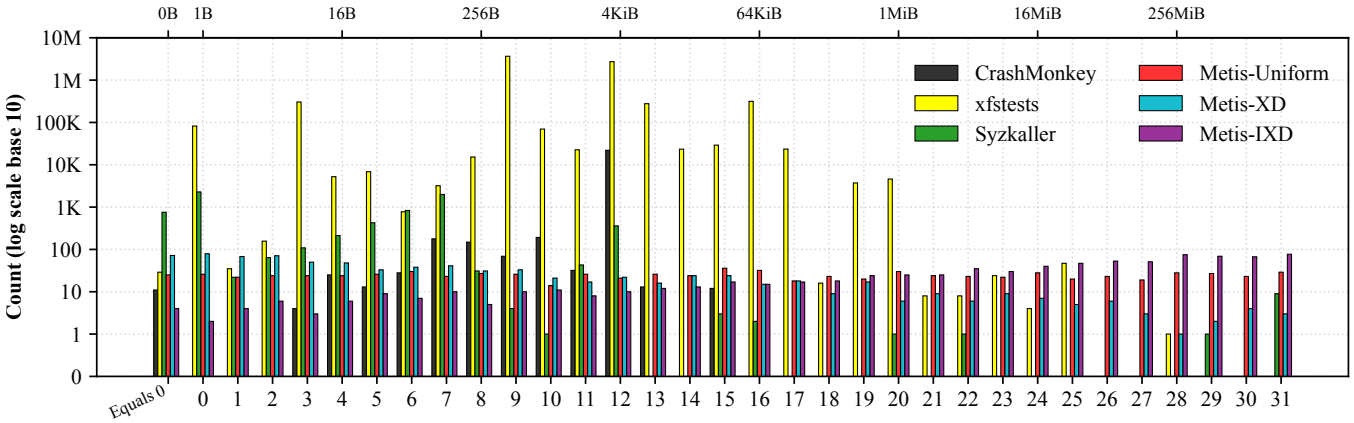

Figure 4: Input coverage (counts, log_{10}, y-axis) of `write` size (in bytes) for CrashMonkey, xfstests, Syzkaller, and Metis with three different weight distributions. The x-axis denotes the power of 2 of the write size (shown as x2-axis). Note a special "Equals 0" x-axis value for writes of size zero.

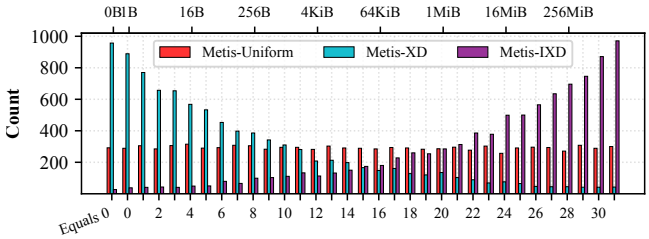

Figure 5: Input coverage of `write` size (in bytes) for Metis-Uniform, Metis-XD, and Metis-IXD, each running for 4 hours. The x-axis and x2-axis here are the same as in Figure 4, but the y-axis shows counts on a linear scale. As seen, with a longer run, the expected distributions are more accurate.

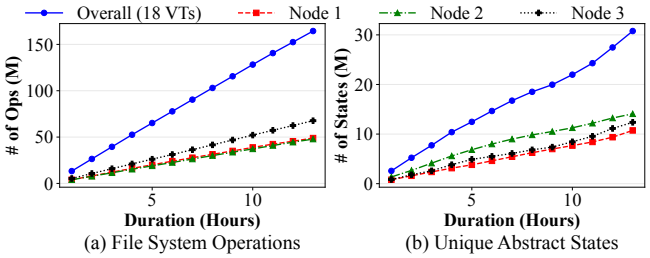

Figure 6: Metis performance with Swarm (distributed) verification, measured in terms of the number of operations and unique abstract states (in millions). Each node runs 6 VTs (one per CPU core), for a total of 18 unique VTs that collectively explored the state space. As seen, performance scales generally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

to be more popular in applications. The probability of each input partition is set to $0.9\times$ smaller than the previous one (in frequency order); all probabilities are then normalized to sum to 1.0. Metis-IXD emphasizes the inverse: testing input partitions with larger write sizes, on the hypothesis that they are less used by applications and thus latent bugs may exist. Here, the probability of each test partition is $0.9\times$ that of the next *larger* partition.

In Figure 4, 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; and when we ran it six times longer still (24 hours), the CV fell to a mere 2.6%. Due to space limitations, we omit showing the input coverage for other Metis-supported syscalls.

5.2 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 6 shows the aggregate performance of the six VTs on each node, as well as the overall performance across all 18 VTs. We measured both file system operations (left) and unique abstract states (right).

All VTs exhibited a linear increase in the number of operations executed over time. Over 13 hours, these 18 VTs executed more than 164 million operations, with each VT averaging 195 ops/s.

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

Different VTs might explore the same states, as each VT operates independently and without communicating with others. We evaluated the proportion of states explored by more than one VT, which represents "wasted" effort, a figure we want minimized. Our results showed that only about 1% of all states were duplicated across all VTs. Therefore, the redundancy of states explored by multiple VTs is relatively small and acceptable.

Bug#	File System	Causes & Consequences	Deterministic	Confirmed	New Bug
1	BetrFS [36]	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	✔	✘	✔

*Table 2: Kernel file system bugs discovered by Metis. 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.*

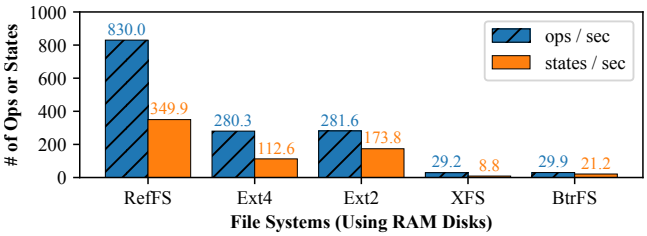

Figure 7: Performance comparison between RefFS and other mature file systems while being checked by Metis. The y-axis applies to both ops/sec and states/sec.

5.3 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 7 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 (§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 bugs in RefFS, xfstests often misreported the bug information. For example, although we implemented RefFS's `link` operation, it still did not pass generic test #2, incorrectly indicating that the operation was unsupported. For that reason, we also used Metis to check RefFS with Ext4 as the reference. We discovered and fixed 11 RefFS bugs, aided by Metis's logs and replayer. Those bugs included failure to invalidate caches, inaccurate file size updates, erroneous `ENOENT` handling, and improper updates to `nlink`, among others. After fixing them, we evaluated RefFS against Ext4 using 18 distributed Metis VTs for 30 days, executing over 3.1 billion operations and exploring 219 million unique states. No discrepancies were reported, demonstrating that RefFS's reliability and robustness are similar to Ext4's—but with better performance when used as Metis's reference file system.

5.4 Bug Finding

With RefFS as our reference file system, we applied Metis to check seven existing file systems: BetrFS [36], BtrFS [68], F2FS [49], JFFS2 [80], JFS [35], NILFS2 [18], and XFS [82], discovering potential bugs in five. Table 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 [71] or Syzkaller [25]. Metis identified an F2FS bug that was not detected by Hydra [43]. We also checked file systems (*e.g.*, BetrFS) that are not currently supported by Hydra [43].

We found bugs using Metis through different indicators. Discrepancies reported by the differential checker accounted for seven out of twelve detected bugs (# 2–6, 9, and 11). The remaining five 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.

FS Testing Approach (Examples)	Input Versatility	Effort to test new FS	Effort to add new ops	State Tracking	Code Coverage Tracking	Bug Detection
Metis: this work	👍👍👍👍	👎	👎	✔	✘	Behavioral discrepancies
Traditional Model Checking: CVFS [21], CREFS [88]	👍	👎👎👎👎	👎👎👎👎	✔	✘	User-specified assertions
Implementation-level Model Checking: FiSC [87], eXplode [86]	👍	👎👎👎	👎👎👎	✔	✘	User-written checkers
Fuzzing: Syzkaller [25], Hydra [43]	👍👍👍	👎👎👎	👎	✘	✔	External checkers
Regression Testing: xfstests [71], LTP [58]	👍	👎👎👎	👎👎👎👎	✘	✘	Preset expected outcome
Automatic Test Generation: CrashMonkey [59], Dogfood [12]	👍👍👍	👎	👎👎👎	✘	✘	External checkers or an oracle

Table 3: Comparison of representative file system testing tools. In column 2, the more 👍 symbols, the more relatively versatile the system is; conversely, in columns 3–4, more 👎 symbols denote more effort.

Of the remaining unconfirmed bugs, four were deterministic and three 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).

6 Related Work

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 3 summarizes these approaches across various dimensions.

Traditional model checking [21, 88] 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 [53].

Implementation-level model checking [86, 87] 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 [86, 87] or manually crafting system-specific checkers [86]. Additionally, existing work [86, 87] 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 [86]. Moreover, other model-checking approaches rely on fixed test inputs [21, 86] and lack the versatility to accommodate different input patterns. All model-checking approaches, including Metis, track file system states to guarantee thorough state exploration [15], a feature often lacking in other approaches.

Model checking and fuzzing are orthogonal approaches, each with its own advantages and disadvantages. File system fuzzing [25, 43, 83, 85] 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 [43, 85] also corrupt metadata to trigger crashes more easily and use library OS [65] 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 [89] 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 [24] (memory errors) and SibylFS [67] (POSIX violations). Current fuzzers configure the tested syscalls but not their arguments [25,70], 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 [71] and LTP [58] 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 [12, 47, 59] creates rule-based syscall workloads (*e.g.*, opening a file before writing) and employs external checkers (*e.g.*, KASan [24]) or an oracle [59] 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 [59] exclusively detects crash-consistency bugs. We do not include a comparative analysis of testing for other storage systems, such as NVM libraries [19] and data structures [20], 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 [14] uses an extended Hoare logic to define a crash-safe specification and avoid crash-consistency bugs. Yggdrasil [72] constructs file systems that incorporate automated verification for crash correctness. DFSCQ [13] introduces a metadata-prefix specification to specify the properties of `fsync` and `fdatasync` for avoiding application-level bugs. SFSCQ [34] 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 [13, 14, 72] or concurrency [93]), 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 [43]). 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.

7 Conclusion

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

In this paper, we presented Metis, a versatile model-checking framework that can thoroughly explore file-system inputs and states. Metis abstracts file-system states into a representation that can be used to compare the file system under test against a reference one. We designed and built RefFS, a reference POSIX file system with novel features that accelerate the model-checking process. When used with Metis, RefFS is 3–28× faster than other, more established, file systems. We extensively evaluated Metis's input and state coverage, scalability, and performance. Metis, helped by RefFS, can speed file-system development: we already found a dozen bugs across several file systems. Overall, we believe that Metis, with its unique features, serves as a valuable addition to file system developers' tool suite. Finally, Metis's framework is versatile enough to be adapted to other systems (*e.g.*, databases).

Future work. Our near-term plans include expanded state exploration using Swarm verification, investigating any bugs we discover, and then fixing and reporting them. We are also beginning to test network and distributed/parallel file systems [29].

In the long run, we plan the following: (i) Metis can trigger nondeterministic bugs, such as race conditions. Therefore, we need to integrate techniques to more deterministically explore and reproduce such bugs [23]. Also, we plan to explore kernel thread interleaving states to find more concurrency bugs [83]. (ii) We intend to enhance Metis by emulating crash states to identify crash-consistency bugs in kernel file systems [47, 59]. (iii) We aim to add support for testing controlled file-system corruptions [29, 85]. For example, if both RefFS and the test file system can be corrupted in a logically identical fashion, Metis can investigate more error paths (*e.g.*, those leading to `EIO`).

Acknowledgments

We thank the anonymous FAST reviewers, our shepherd Haryadi S. Gunawi, and Dongyoon Lee for their valuable comments; and to Yizheng Jiao and Richard Weinberger for their assistance in confirming bugs in BetrFS and JFFS2. We also thank fellow students Rohan Bansal, Tejeshwar Gurram, and Shushanth Madhubalan for their contributions. This work was made possible in part thanks to Dell-EMC, NetApp, Facebook, and IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-1900589, CNS-1900706, CCF-1918225, CNS-1951880, CNS-2106263, CNS-2106434, CNS-2214980, CPS-1446832, ITE-2040599, and ITE-2134840.

A Artifact Appendix

Abstract

The paper artifact contains the implementations of the *Metis* model-checking framework, the *RefFS* reference file system, and other necessary components as well as the code needed to reproduce most of the experimental results presented in this paper. Our artifact allows straightforward checking of those Linux file systems supported by Metis, and can be easily adapted to examine other file systems. We also provide documentation that explains how to set up the environment, scale up the exploration process, and detect and reproduce file system bugs based on Metis's logs and replayer.

Scope

This artifact is intended not only to validate the main claims in this paper but also to enable others to use and extend our tools, find more file-system defects, and enable future research. Specifically, we include code that automatically reproduces the results discussed in §5, including:

- Input coverage results shown in Figures 3, 4, and 5.
- Metis performance using Swarm verification in terms of operations and unique abstract states per second, as presented in Figure 6.
- RefFS performance compared to other file systems while using Metis, as shown in Figure 7.
- Detection and reproduction of file system bugs that were found by Metis.

Contents

The artifact includes two main Git repositories: the Metis file system model-checking framework and the RefFS user-space reference file system. Additionally, it contains several auxiliary Git repositories that support a basic model-checking facility and coverage analysis. Specifically, the artifact includes:

- Source to compile and execute the Metis framework for checking file systems.
- Source to build and operate the RefFS reference file system.
- Scripts to reproduce most of the experimental results appearing in this paper.
- Modified SPIN and Swarm verification scripts, optimized for seamless integration with Metis.
- The IOCov [52] tool used to compute input and output coverage for file-system testing tools.

Hosting

All the repositories are hosted on GitHub with README files for documentation; some are archived using Chameleon Cloud's Trovi service [41] and Zenodo with a permanent DOI.

Metis Repository

- Repository: *https://github.com/sbu-fsl/Metis*
- Branch: "master"
- Commit: ae08f6802be7cacb614847ebce78c18af86d553a
- Zenodo Archive [50]: *https://zenodo.org/records/10537199*
- DOI: *https://doi.org/10.5281/zenodo.10537199*

RefFS Repository

- Repository: *https://github.com/sbu-fsl/RefFS*
- Branch: "master"
- Commit: 680f5539791fc9c410d7d3cfcf2970ec4edf43a6
- Zenodo Archive [51]: *https://zenodo.org/records/10558327*
- DOI: *https://doi.org/10.5281/zenodo.10558327*

Other Repositories

- Repository of the Modified SPIN: *https://github.com/sbu-fsl/fsl-spin*
- Repository of the Modified Swarm Verification Tool: *https://github.com/sbu-fsl/swarm-mcfs*
- IOCov Repository: *https://github.com/sbu-fsl/IOCov*

Requirements

Generic Requirements

The artifact requires x86 Ubuntu 20.04 or 22.04 with one of the following Linux kernel versions: 5.4.0, 5.15.0, 5.19.7, 6.0.6, 6.2.12, 6.3.0, or 6.6.1. It may work with other Linux distributions and kernels but we did not test that.

Metis is both CPU- and memory-intensive. Running the artifact does not demand specific CPU resources, but a higher-end CPU can improve the performance of Metis's state-space exploration. Metis's memory usage depends on the type of file system being checked. Generally, the required memory size needs to be at least the sum of the minimum mountable sizes of the two file systems being compared (the file system under test and a reference file system), multiplied by Metis's maximum search width (default 10,000). Therefore, larger amounts of RAM are helpful. If sufficient RAM is not available, we recommend setting up a swap disk on a fast device such as a high-end SATA-SSD or NVMe-SSD. Metis also generates many logs during execution, so we recommend using at least a 500GB disk to avoid running out of log space.

This artifact comes with several prerequisites. We therefore provide a script `script/setup-deps.sh` in the Metis repository to automatically install all the required tools and libraries on an Ubuntu system.

Requirements for running Metis with Swarm verification

When using multiple parallel Verification Tasks (VTs) in Metis, the required computational resources amount to the demand of a single VT, multiplied by the total number of VTs. Specifically, the number of CPU cores should equal or exceed the number of VTs operating on a machine. Similarly, memory and disk resources should linearly scale with the number of VTs. The number of VTs can be configured in the `fs-state/swarm.lib` file within the Metis repository.

When VTs in Metis are distributed over multiple machines, each machine must be equipped with resources proportional to the number of VTs it runs. Moreover, in this distributed setting, one machine should be designated as the primary, with the remaining machines serving as workers. The primary machine should be set up for password-less SSH key-based access to the workers. We recommend that the hostnames of the workers are accurately entered in the `swarm.lib` configuration file on the primary machine.

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

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

[59] 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.

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

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

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

[67] 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.

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

[69] 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.

[70] 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.

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

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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/.

[77] 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.

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

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

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

[81] 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.

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

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

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

[90] 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.

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

[92] 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.

[93] 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.