SnapCC: Effective File System Consistency Testing Using Systematic State Exploration

JIANZHONG LIU, YUHENG SHEN, YIRU XU, HAO SUN, and YU JIANG*, Tsinghua University, China

Modern file systems have become increasingly feature-rich and highly complex, making crash consistency increasingly difficult to perform correctly. Thoroughly testing file systems for crash consistency bugs, however, is difficult to achieve good results due to insufficient state exploration, a lack of guidance for test case generation, and missing support for modern file system features.

In this paper, we present a new approach towards testing file system consistency: systematic file system persistent state exploration. In contrast to previous efforts, our design addresses these shortcomings through testing the crash consistency property of file systems *systematically* using the following procedures. Initially, we use system call generation and execution feedback from fuzzers to generate workloads that stress the file system code. During this process, we systematically explore all possible persistent states of the underlying file system for the given workload, and subsequently use them as file system image inputs for the crash recovery routines to produce a corresponding file system state. After the file system finishes processing an image input, we deploy an efficient file system checker to compare the contents of the image to that of a correct image and determine whether the image is inconsistent, consequently determining whether we has triggered a crash consistency bug in the underlying file system.

We implement a prototype tool SnapCC and deployed it for testing multiple mainstream file systems on Linux. We compared its effectiveness along with other relevant tools Hydra and B3, where our results show that SnapCC achieves 16% to 44% better coverage over Hydra, and finds 15 new consistency bugs, whereas B3 and Hydra finds 2 and 6, over a period of 2 weeks, further demonstrating SnapCC's effectiveness in discovering file system consistency bugs. To demonstrate our approach's adaptability, we also tested SnapCC on 5 other file systems, upon which 7 additional bugs were found.

CCS Concepts: • **Security and privacy** → **File system security**; **Operating systems security**; *Database and storage security*; *Software and application security*; • **Computer systems organization** → Dependable and fault-tolerant systems and networks.

Additional Key Words and Phrases: file system crash consistency testing, file system fuzzing, kernel security, systems security

ACM Reference Format:
Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. 2025. SnapCC: Effective File System Consistency Testing Using Systematic State Exploration. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 24 pages. https://doi.org/XXXXXXX.XXXXXXX

*Yu Jiang is the corresponding author

Authors' Contact Information: Jianzhong Liu, liujz21@mails.tsinghua.edu.cn; Yuheng Shen, shenyh20@mails.tsinghua.edu.cn; Yiru Xu, xuyr21@mails.tsinghua.edu.cn; Hao Sun, sunhao.th@gmail.com; Yu Jiang, jiangyu198964@126.com, School of Software, Tsinghua University, Beijing, China.

ACM 1557-7392/2025/1-ART1
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

File systems, unlike many ephemeral programs, need to explicitly write data onto blocks on non-volatile storage devices for it to persist across power cycles. In an optimal setting, file systems need not handle cases where a drive or partition for a given file system is *inconsistent*, where data on disk is not in a valid state. However, in reality, accidents such as power interruptions, drive failures, and possibly human errors happen, such that they can significantly interrupt the persistence operations of a file system. Therefore, file system designs need to accommodate for circumstances where data persistence operations may fail midway, leaving an inconsistent file system on disk, thus requiring certain procedures to recover the data on disk into a valid state for further operations to proceed.

File system designs have developed ideas and methodologies, including log structured file systems, to combat such problems. However, as modern file systems have evolved significantly over their predecessors, where they adopt many new features (e.g. Linux's BTRFS [27] and FreeBSD's OpenZFS incorporate advanced features such as compression, encryption, snapshots, etc.), the addition of such features significantly increases the complexity involved with these file systems when implementing persistence operations and consequent crash recovery procedures. This significantly increases the difficulty involved in designing robust crash recovery procedures for modern file systems, where insufficient examination and testing may lead to critical data-losing bugs living in shipped software.

In general, testing is an effective means of rooting out concrete bugs in program code. Specifically for testing file systems for crash consistency bugs, the procedure mainly consists of the following steps: 1) a block device with a file system is mounted as the target; 2) the tester procures a file system invocation list containing calls to read and write from the target file system; 3) the tester invokes the relevant invocations on the mounted file system; 4) the subsequent file system is checked for any errors. Previous research, such as B3 [22] and Hydra [15], tackle the issue of testing file systems for consistency bugs. B3's approach enumerates all possible combinations of POSIX file system operations with a length less or equal to 3 on a target file system, and injects a fault after an appending persistence operation (usually a *sync()* call) to trigger possible crash consistency bugs. Hydra uses fuzzing as a means to explore a greater input space by using feedback-guidance to generate test cases of file system invocations,

While their approaches have found concrete crash consistency bugs, their main focus is on POSIX-compliant file system operations while performing sequence-wise fault injection. With modern file systems, we find that file system crash consistency bugs can also occur during design-specific operations, and their interactions with POSIX file system operations, such as experiencing a fault during snapshot creation, etc. To effectively find such instances of crash consistency bugs through testing, we need effective means of test case generation and crash consistency bug detection. Specifically, the proposed test case generation methods need to create diverse on-disk file system images produced by triggering an abundance of file-system-specific operations that effectively explore the feature space of the file systems. Additionally, the crash consistency bug detection mechanisms need to effectively find all correct states of the file system as testing oracles to determine exceptional states as potential bugs.

Consequently, to test file systems for crash consistency bugs with better effectiveness, file system crash consistency checkers need to address the following challenges. First, generating file system invocations to trigger diverse functionalities in the file systems is difficult to accomplish, considering the different interfaces and semantics of such invocations between different file systems. Additionally, file systems tend to write data to the disk asynchronously, thus making it significantly difficult to capture all possible on-disk states generated through executing a file system invocation sequence to use as test cases. Finally, detecting crash consistency bugs post-execution is difficult as file systems are diverse in their interpretations and implementations of crash consistency, whereas manual inspection is undesirable as the number of test cases executed is huge.

We observe that in reality, both systematically exploring possible on-disk states for each invocation sequence and automatically finding possible valid states are two sides of the same coin: persistence operations consist of multiple write operations to the block device for updating the relevant metadata and data blocks, thus through enumerating through all states of the underlying storage device during the execution of an invocation sequence, we are in essence finding all possible states that the device is in under any failure, while finding the valid states can be achieved in identifying *safe* states after every persistence point, such as executing *sync()*, as safely unmounting the corresponding block device results in a consistent file system structure. Furthermore, to address the invocation sequence generation and mutation problem, we can borrow established methods from kernel fuzzing though utilizing *expert written* system call specifications for kernel fuzzers, such as Syzkaller [32], and prioritizing file system modification operations, including file-system-specific operations.

Using these observations, we propose SnapCC, a file system crash consistency bug detection tool that performs systematic file system state exploration to greatly increase the effectiveness in rooting out crash consistency bugs. SnapCC uses kernel fuzzing methods to generate invocations to file systems, prioritizing file-system-specific operations, while designs custom modules and routines to generate test cases and find valid states from the execution of invocations, and consecutively uses such to find and detect crash consistency bugs. Specifically, to effectively detect consistency bugs in file systems, SnapCC mainly utilizes a specification-based invocation sequence generator/mutator, an Automatic Valid State Finder to generate test oracles, a Systematic On-Disk State Explorer to produce all possible on-disk states for a given file system invocation sequence, and finally a Consistency Verifier to identify instances where the file system exhibits an inconsistency bug. SnapCC leverages existing specifications for file-system-relevant system calls for effective input generation and mutation, which are extracted from Syzkaller and further improved with more file-system-specific system calls. For systematic exploration of possible crash states and automated testing oracle generation, SnapCC leverages snapshots to rapidly iterate through persistent states during an invocation process. To identify valid persistence states, SnapCC first identifies persistent points within the invocation, and acquires a snapshot of such a valid state after safely unmounting after each persistence point. For systematic exploration, SnapCC hooks QEMU's block device driver to identify each write operation initiated by the file system. When such a write operation commences, SnapCC takes a snapshot of the block device backing file during and after the write operation. After concluding an invocation sequence, SnapCC mounts each snapshot of the block device during execution, and compares its state to the set of valid states. If no match is found, then SnapCC reports such a case as a crash consistency bug.

We implemented SnapCC and evaluated its effectiveness with B3 (implemented as two tools ACE+CrashMonkey [23]) and Hydra as comparison tools. Our results on Linux file systems BTRFS, F2FS, and ext4 show that SnapCC achieves a 16% to 44% improvement in coverage over Hydra. For bug finding capabilities, we first used a baseline dataset consisting of the historical crash consistency bugs found by B3 and Hydra to test SnapCC's bug finding proficiency, where it successfully found all bugs in the dataset. On BTRFS, F2FS and ext4, SnapCC found 15 new consistency bugs over a period of 14 days, where B3 found 2 and Hydra found 6. SnapCC's adaptability to new file systems can be demonstrated by its test run on the file systems UFS, XFS, BCacheFS and OpenZFS, where it also found 7 new crash consistency bugs. We also examined the overhead of SnapCC's systematic state exploration capabilities, which show that SnapCC's execution time is at most 2× than that of Hydra, which is acceptable considering SnapCC's greater effectiveness in uncovering crash consistency bugs. Additionally, our ablation tests also show that, using the same set of POSIX file system operations, SnapCC still achieves a 22% increase in code coverage over B3 and Hydra, demonstrating the effectiveness of SnapCC's approach in finding cases of crash consistency bugs in ordinary POSIX operations.

In conclusion, our main contributions in this paper are given as follows:

- We identify that modern file system have implementation-specific features and functionalities that impact crash consistency correctness, thus requiring more systematic and versatile crash consistency testing methods to generate more effective file system invocation sequences, identify more unique file system on-disk states as the test cases, and automated valid state finding for multiple file systems.
- We propose SnapCC, a file system consistency testing tool that addresses the aforementioned issues with effective, specification-driven invocation sequence generation and mutation, systematic file system on-disk state exploration for test case generation, automated valid-state-iteration-based testing oracle generation, and crash consistency bug detection mechanisms.
- We evaluated SnapCC against previous research works Hydra and B3 on Linux file systems BTRFS, F2FS and ext4. Our tests show that SnapCC achieves a 16% to 44% improvement in coverage compared to Hydra, and additionally SnapCC found 15 new bugs in various file systems over the course of 14 days, whereas B3 and Hydra found 2 and 6, respectively. Furthermore, our analysis of SnapCC's execution overhead show that SnapCC's execution throughput is comparable to state-of-the-art file system fuzzing techniques, and our ablation tests show that SnapCC still achieves a 22% increase in code coverage over B3 and Hydra even when using only POSIX-compliant file system invocations. Adaptation tests also extend SnapCC's scope to four other file systems, where SnapCC found 7 new crash consistency bugs.

2 Background

2.1 File System Crash Consistency

File system crash consistency is the ability of a file system to keep its internal data structures and corresponding user data valid during unexpected failures such as power outages. In the early days of computing, file systems required simple consistency checking measures, as all system calls were executed sequentially, including disk I/O, thus file systems is in constant consistency with its backing store. However, as the performance of computers grew exponentially, the speed disparity between memory and persistent storage grew significantly, necessitating separation of the in-memory file system data structures and the on-disk file system contents. Immediate changes to the file system's data structures remained in memory for efficiencies, where explicit persistence calls (*sync()*, *fsync()*, etc. in POSIX) or kernel maintenance threads will flush the *dirty* contents, i.e. those that have diverged from disk contents, into the backing store to persist the modified data. In the event of a persistence operation failure due to power failures, device malfunctions, operating system crashes, etc., the file system should recover from such failures to a *valid state*, i.e. a state where the internal data structures and data of the file system are coherent, without lost or corrupted data. This requires meticulous attention to the design and implementation of persistence and recovery routines, where a single error can lead to disastrous results.

Modern file systems have introduced many quality-of-life feature enhancements, such as volume and *subvolume* management, snapshots, file and data de-duplication, etc., which, while presenting the end user with a rich feature set, impose significant difficulties for designing correct crash recovery routines. Intuitively speaking, the more complex that the file system's functionalities become, the more convoluted and error-prone that writing crash recovery routines become, potentially having more bugs within its code.

There is much work in both academia and industry that strives to pin-point and consequently eradicate consistency bugs from file systems, both from a constructive design perspective and post-implementation testing perspective. For the former, many file systems use techniques such as journaling or transaction-based operations which both allow the file system to undo unfinished operations during the recovery from a crash, therefore maintaining the file system's consistency. However, practical testing tools that find consistency bugs within file systems demonstrate that bugs within the file systems' implementations may still result in inconsistent states after crashes.

Currently, state-of-the-art file system consistency dynamic testing tools include B3 and Hydra. B3 is a black-box approach that tackles the state exploration problem through exhaustively searching through possible combinations of POSIX file system operations with a length less or equal to three. It then executes each found sequence on a file system image with a persistence system call *sync()* appended to initiate the file system persistence operation. After *sync()* returns on each execution, B3 simulates a crash at a random time point. It then remounts the file system image and compares the contents to the expected results when a program calls *sync()*. In its implementation, B3 consists of two tools, namely ACE and CrashMonkey, where the former tools is tasked with generating all possible calls, whereas the latter executes the generated traces, simulates crashes, and triggers consistency validation routines. Hydra is a fuzzing framework proposed by Kim et al. [16] which integrates many file system bug finding capabilities, including consistency checking. In contrast to B3, Hydra does not aim to exhaustively check all possible combinations with a length upper bound; rather it explores an infinite input space through randomized input generation and mutation techniques. Hydra also sports a full fuzzing feedback loop, allowing it to preserve interesting inputs for further mutation and testing. Hydra's consistency checking component, *SymC3*, is an emulation-based tool that simulates the file system operations for a given invocation sequence. It produces valid states for an invocation sequence and compares the file system image after remounting and allowing the file system to recover after a simulated crash.

2.2 Fuzzing

Fuzzing is a popular and effective dynamic testing technique that has been applied to testing various types of software with success. In principle, fuzzing repeatedly tests a program through feeding its generated or mutated inputs and observing for any exceptional behavior. Many state-of-the-art fuzzers (fuzzing tools) use greybox feedback to assist input generation and mutation.

Kernel fuzzing adapts fuzzing to testing operating system kernels for bugs. The kernel under test runs in a virtualized environment, where an agent program, which delivers fuzzer-generated inputs into the kernel. Current state-of-the-art kernel fuzzers feed input data to the kernel through invoking various system calls. The specific system calls to call and their respective arguments are called a system call sequence. To generate such inputs, kernel fuzzers generally use *system call specifications* to generate new calls and mutate existing system call sequences.

Syzkaller, a state-of-the-art kernel fuzzer, utilizes a plethora of system call descriptions written by kernel experts to generate a wide assortment of system call sequences. The descriptions consist of concrete system call declarations, relevant data types and their aliases, input or output directions for arguments, as well as constants, such as flags. For certain system calls that take certain arguments or exhibit extraordinary behavior when used with a certain argument, Syzkaller uses system call specializations to specify these special system calls. We show some sample specifications in Listing 1. In the listing, the first line shows a declaration of a resource type, which represents entities passed in and out of system call invocations, and the subsequent lines show specifications for specific system calls. For instance, the second and third lines are for the generic *open()* and *mount()* system calls, whereas the last line is for a specialized *ioctl()* system call, i.e. an *ioctl()* call that describes an invocation to the F2FS file system for defragmentation operations.

```
1 resource fd[int32]: -1
2 mount(src ptr[in, blockdev_filename], dst ptr[in, filename], type ptr[in, string[
      filesystem]], flags flags[mount_flags], data ptr[in, string, opt])
3 open(file ptr[in, filename], flags flags[open_flags], mode flags[open_mode]) fd
4 ioctl$F2FS_IOC_DEFRAGMENT(fd fd, cmd const[F2FS_IOC_DEFRAGMENT], arg ptr[inout,
      f2fs_defragment])
```

Listing 1. Sample Syzkaller system call specifications, including definitions for resources (Line 1), declarations for regular system calls (Lines 2 and 3), and specialized system calls that specify a specific function segment of a general system call (Line 4).

Kernel fuzzers can also partially support file system testing, as the file operations on file systems are conducted through system calls to the kernel. However, compared to specific file system fuzzers such as Hydra, kernel fuzzers are not designed specifically for syntactically-correct and semantically-rich file system invocation sequence generation, and have insufficient checkers for identifying different types of errors outside of memory violations in file systems.

3 Motivation

While previous works such as B3 and Hydra utilize testing to find concrete consistency bugs, their main focus is detection through POSIX-standard file system invocations and their corresponding functionalities However, modern file systems have outgrown their primitive roots, and require additional testing capabilities to find bugs in these file-system-specific code.

While B3 reduces the input space in order to exhaustively search for combinations of file system operations that may produce consistency bugs, the limit on the number of file system operations prohibit its capabilities from finding consistency bugs that require more complex file system invocation sequences from being found, especially considering the complexity of guiding file systems into states corresponding with implementation-specific functionalities. Additionally, B3's consistency testing produces considerable false positives, as its consistency oracle only produces a subset of the possible valid states that a file system image can be in, therefore B3 frequently flags a consistency bug when the image is actually consistent. Furthermore, B3 is a black-box tool, thus it does not leverage feedback to further test file systems for consistency bugs. Finally, B3 does not support invoking file-system-specific operations, and retrofitting it with such support, in combination with these issues, will result in severely degraded effectiveness.

On the other hand, while Hydra improves upon B3 in many aspects, there are still areas in the file system's functionality that it does not cover. First, while Hydra can explore a more high-dimensional input space than B3, it retains its focus on POSIX file system operations, and requires extensive modifications to accommodate code to generate file-system-specific operations for each intended target file system. Additionally, Hydra's approach to simulating faults is similar to that of B3, which only triggers after the execution of an invocation sequence, but does not consider the variety of on-disk states that the file systems' underlying persistence operations may produce. Furthermore, while Hydra's *SymC3*, the verifier that produces all possible valid states, has its implementation constructed manually, with code for further adaptation to different file systems, thus requiring intensive domain knowledge and manual efforts, also with the risk of introducing bugs into the verification process.

To test modern file systems for crash consistency bugs with consideration to implementation-specific features, and more thorough bug detection, we need to address the following challenges. First, to generate diverse payloads to test the file system, we need to systematically explore potential on-disk states for a given invocation sequence during execution, which is difficult to achieve due to the varied interfaces and semantics of different file systems, whereas retrofitting existing tools with such functionalities require significant expert knowledge of the individual file systems and extensive manual effort. Second, as file systems write data to the disk asynchronously, it is difficult to interpret the different on-disk states of the executed invocation sequences, hence hindering the test case generation of on-disk file system images. Previous works including B3 and Hydra take a randomized approach towards finding these states, but this approach misses many more possibilities that can trigger crash consistency bugs. Finally, designing testing oracles to detect crash consistency bugs post-execution is no trivial job, as we need to consider all possible *valid* on-disk states that the file system may be in, which depends on both the specific file system invocations performed and implementation-specific details.

We make some observations that help us address the aforementioned challenges:

First, userspace programs interact with files through system calls, including POSIX-defined file system APIs, such as *creat()*, *chmod()*, *ftruncate()*, etc., whereas modern file systems additionally support more sophisticated functionalities, generally through the use of *ioctl()* with a *request* parameter that defines the actual operation. While pure file system testing tools generally use only POSIX invocations, specification-based input generation kernel fuzzers such as Syzkaller leverage system call descriptions written by file-system-domain experts, including a wealth of specifications to interact with the specific file system's implementation-specific functionalities. Therefore, utilizing such information will assist testing tools in triggering more non-POSIX functionalities more effectively.

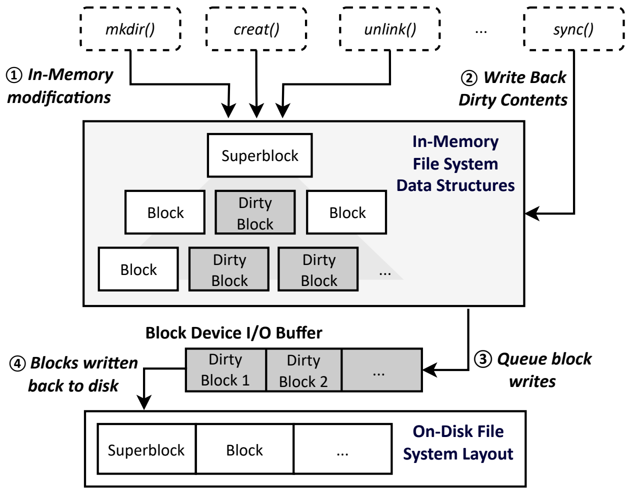


Fig. 1. File system persistence internals in OS file systems. Generally, modifications to the file system are only kept in memory initially, whereas synchronization operations or a routine operation flushes the *dirty*, or modified, blocks to disk. During this process, any failure that results in its interruption may result in inconsistency issues, as metadata or data itself has not been fully written back.

Additionally, we observe that persistence operations of file systems are conducted through a series of writes only after persistence operations are invoked, therefore the space of file system on-disk states can be *systematically* explored. We demonstrate this in Figure 1, where the effects of file system operations such as *mkdir()*, *creat()* and *unlink()* only affect the in-memory data structures of file systems, where the backing store block device is unaffected. Only when an explicit persistence operation is invoked, such as calling the *sync()* system call or a kernel worker thread flushes the data to disk, is the data written back. The persistence operations consist of multiple write operations to update the metadata and contents of files, as well as the file system's own data organizations. Therefore, the problem of traversing the state space of possible on-disk states from a file system invocation sequence can be converted into enumerating all possible states of the persistent storage during the invocation.

Furthermore, we assert that the consistency of file system is held true across persistence operations. This assertion will fail only in the presence of semantic errors or memory bugs, which are topics orthogonal to consistency bugs. Therefore, determining consistent states that the file system can recover to during a crash can be resolved specifically by appending persistence operations directly after every file-system-modifying operation, then safely unmounting the file system, forcing the operating system to flush all dirty pages to storage immediately. We then collect all such states that originate after each modification operation, and thus obtain a set of valid states to recover to for the file system.

4 Design

We now introduce the design of our approach in the form of SnapCC, a file system crash consistency bug fuzzer. The overall architecture of SnapCC is shown in Figure 2. As shown in the diagram, the runtime organization of SnapCC uses a feedback-guided fuzzing loop, with on-disk state generation based on an Invocation Sequence Generator/Mutator (§4.1) and a Systematic On-Disk State Explorer (§4.2), while the specific testing oracle is generated through an Automated Valid State Finder (§4.3) , and finally the Consistency Verifier checks the all found on-disk states with the set of valid states as testing oracles (§4.4).

The overall workflow is given as follows. At the start of each iteration, SnapCC first produces a file system invocation sequence, either through seed sequence mutation or through generating a new sequence using specification-based generation methods. The generated file system invocation sequence is then used by the Systematic On-Disk State Explorer to generate all possible on-disk states after a fault, which make up the set of input payloads to be tested on the file system. The same sequence is also sent to the Systematic On-Disk State Explorer, which run the sequence to produce a set of valid system states as the testing oracle. Subsequently, these sets of states are then verified by the Consistency Verifier, which mounts all possible file system states, waits for the file system to recover to a fixed state, and then attempts to find a match to one of the valid file system states, upon which if no match is found, then SnapCC determines that a crash consistency bug has been found. The following sections will discuss the designs of SnapCC's components in detail.

4.1 File System Invocation Sequence Generation

To generate diverse on-disk states through triggering file system persistence operations, we need to generate syntactically correct and semantically rich file system invocation sequences. File system invocation sequences consist of a file-system-relevant system call sequence and an base file system image. Upon each iteration during testing, SnapCC generates such a system call sequence and file system image to manipulate the underlying target file system into activating different persistence operations under different contexts. As aforementioned, SnapCC can either mutate an existing file system invocation sequence, or generate a new invocation sequence through system call descriptions. The generation process is outlined in Figure 3. The following sections cover the generation process of both file system images and system call sequences during either seed mutation or input generation:

File System Invocation Generation: For generation from scratch, SnapCC finds a base file-system-relevant system call to generate, which is directed towards the target file system's directory or files, and sequentially finds dependent system calls through argument dependencies or system call relations, all expert-written system call specifications. Specifically for file-system-specific

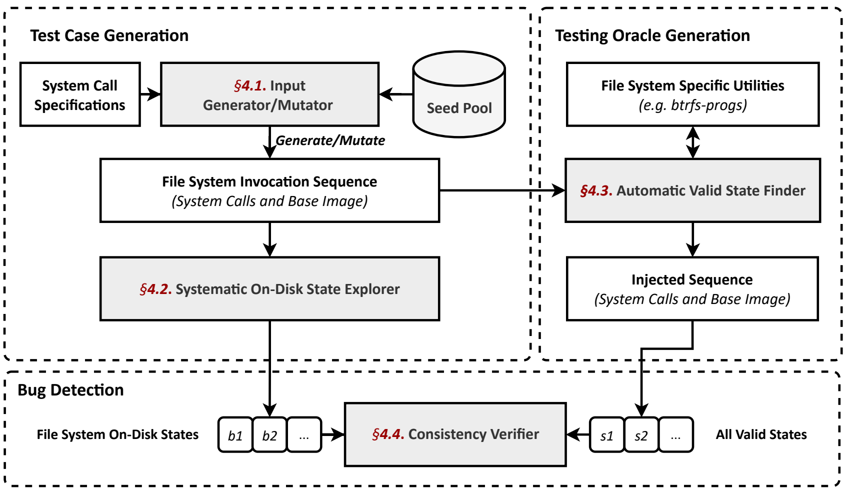

Test Case Generation
System Call Specifications
§4.1. Input Generator/Mutator
Seed Pool
Generate/Mutate
File System Invocation Sequence (System Calls and Base Image)
§4.2. Systematic On-Disk State Explorer
Testing Oracle Generation
File System Specific Utilities (e.g. btrfs-progs)
§4.3. Automatic Valid State Finder
Injected Sequence (System Calls and Base Image)
Bug Detection
File System On-Disk States
b1
b2
...
§4.4. Consistency Verifier
s1
s2
...
All Valid States

Fig. 2. Workflow Diagram of SnapCC. The corresponding sections that detail the inner workings of each component are labeled accordingly. SnapCC generates test cases, as file system on-disk states, to test the whether the file system's crash recovery process can preserve consistency. These on-disk states are created through systematically exploring all possible states (§4.2) triggered through executing the generated file system invocation sequence (§4.1). SnapCC additionally finds all valid states as the testing oracle (§4.3) to verify the set of found states against (§4.4)

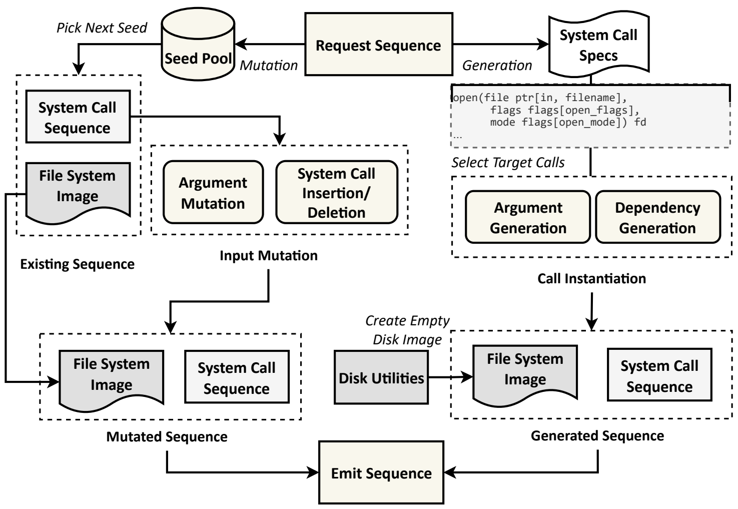

```
open(file ptr[in, filename],
      flags flags[open_flags],
      mode flags[open_mode]) fd
...
```

Fig. 3. SnapCC's sequence generation process. When SnapCC requires a new sequence, a random number generator (RNG in the figure) determines whether to generate a new input from the system call specifications or to mutate an existing input in the seed pool. For the former, SnapCC selects the target system calls from the specifications, instantiates the calls with arguments and dependencies, and produces a system call sequence, which is paired with a generated file system image and emitted for execution. For the latter, SnapCC retrieves the next seed to be tested, mutates the system call sequence, including argument mutation and insertion or deletion of system calls, then emits the resulting sequence.

functionalities, SnapCC also emphasizes calls to trigger these functions, which usually take the form of *ioctl()* system calls with specific arguments. For instance, for BTRFS to trigger a snapshot creation operation, the fuzzer prepends a system call to *ioctl()* with `BTRFS_IOC_SNAP_CREATE` as its command argument.

When producing a sequence through seed mutation, SnapCC first retrieves a seed from the pool, then performs random operations, such as randomly appending a new system call with arguments generated in regards to the previous resource restraints, and/or mutates certain parameters of existing calls in the sequence. Additionally, the mutation process can also choose to *truncate* a seed sequence, where the image contents are updated to the results of the existing invocations, where the invocations are then removed and regenerated. The process of generating a new input based on either specification-based generation or seed mutation is demonstrated in Figure 3.

Image Generation: SnapCC generates new valid file system images through the use of disk maintenance tools bundled with common Linux distributions to avoid generating new images with allocated files or data, and that all on-disk data can be created with valid, canonical file system operations. When input mutation is selected, we use an intermediary on-disk state that corresponds to the starting state of the sequence.

4.2 Systematic On-Disk State Explorer

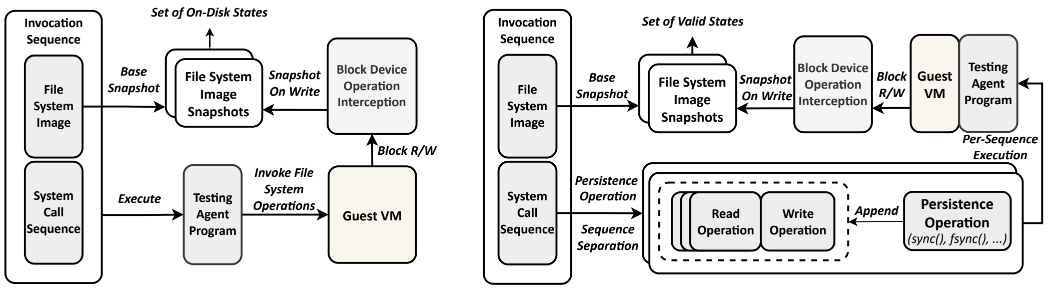
With the file system invocation sequence generated, we wish to generate test cases of file system images that extensively represent the on-disk states where faults may happen, and thus thoroughly test the file systems' consistency maintenance and recovery routines, and therefore expose any bugs through the results of the recovery process. To do so, we use a *Systematic On-Disk State Explorer* to enumerate all possible states that the underlying file system image can be in after a fault during the execution of the file system invocation sequence.

Our observation is that for file systems, a persistence operation to disk involves multiple block device write operations, with each operation organized as a write sequence of blocks, allowing us to fully reach the on-disk file system states by iterating through each modification operation. To do so that allows for versatility across different file systems, we hook and intercept the underlying write operation routine in the guest machine's block device driver. Thus when the block device receives a block-device-level write command, we take two snapshots of the file system image, one during the write, and another after the write operation has finished. The snapshots taken during the execution of an invocation sequence emulate a crash at the specific time of the snapshot, specifically at the end of a persistence operation and in the middle of a write operation.

After executing the entire sequence, we are presented with the possible on-disk states of the file system during the execution. The file system on-disk states are the test cases that are then sent to the Consistency Verifier for verification of crash consistency. This process is depicted in Figure 4a.

4.3 Automatic Valid State Finder

As the valid states that the file system can be in depends on the actual invocations executed on the file system, we need to adaptively identify all possible valid states for a given sequence such that we can identify invalid, and thus inconsistent states. In parallel to finding all possible on-disk states, SnapCC uses the generated file system invocation sequence and determines which *consistent* states that the file system can be in after a recovery from a crash. Since all persistence operations, when completed, should maintain a file system's consistency property, we find all *write*-related operations to the file system as possible persistence operation triggers. Using this as a basis, we perform the following operations the invocation sequence to produce a set of valid states.



(a) The Systematic On-Disk State Explorer. (b) The Automatic Valid State Finder.

Fig. 4. Snapshot-driven exploration of either possible or valid on-disk file system states. Figure 4a shows SnapCC's process in determining the set of possible file system on-disk states, while Figure 4b shows how SnapCC finds all valid states of the file system image. Both methods take advantage of snapshotting for fast iteration of different on-disk states.

4.3.1 Persistence Operation Sequence Separation. First, we append each file system modification invocation with a *sync()* system call. These invocations may include data modifying operations, such as *write()*, or purely metadata manipulation operations, such as some *chmod()*. Then, we consider all such sub-sequences that start with consecutive non-modifying operations, such as *read()*, then subsequently one modification operation, then consecutive synchronization operations, such as *sync()* and *fsync()*. For instance, in the sequence consisting of *open()*, *mkdir()*, *creat()*, *fsync()*, *rename()*, *stat()*, *ioctl()*, *sync()*, we first append *sync()* to all file-system-modifying operations; then, we separate the sequence according to persistence operations; we emerge with the following sequence: (*open()*, *mkdir()*, *sync()*), (creat(), sync()), (fsync()), (rename(), sync()), (stat(), ioctl(), sync()). We denote such sequences as *Persistence Operation Sequences*. These sequences, after executed, can represent valid states, as their contents have been explicitly persisted to the backing store block device. When executed in order and fully committed to disk, these *persistence operation sequences* will produce the valid states that the file system can be in during the original system call sequence's execution.

4.3.2 Enumerating All Possible Valid States. Through the process of finding all persistence operation sequences above, we obtain valid file system states for a given invocation sequence. First, we mount the original sequence's initial file system image, and produce all *persistence operation sequences* from the given system call sequence. Then, we sequentially execute each persistence operation sequence in order. After each run, we safely unmount the file system image and take a snapshot of the resulting file system image. The set of snapshots taken during this process, as they are all valid states for the target file system, and we have exhaustively produced all possible states, is considered the overall set of valid states for a given file system invocation sequence. After this, it is provided to the Consistency Verifier for finding crash consistency bugs during this run. This process is depicted in Figure 4b.

4.4 Consistency Verifier

To verify if on-disk states produced by the file system invocation sequence's execution can expose a crash consistency bug in the file system, SnapCC employs a consistency verifier to perform such validation operations.

First, the Verifier uses a fresh snapshot of the operating system and attaches each on-disk state. Then, it performs a file-system-specific consistency checker run, such as running *fsck.ext4*. This is required as some file systems, such as OpenZFS, explicitly state that consistency is not guaranteed in the event of a failure, thus requiring the execution of its scrubbing utility.

After the file system has finished recovering, we mount the on-disk state. We then mount all valid states in order and compare the contents of the disk in both disk images. The comparison involves the following items: directory tree traversal, individual file metadata comparison, and file contents comparison. To accelerate the comparison process, we compare the hash of files' contents using the cryptographically secure SHA-256 hashing algorithm. Additionally, we also use file system utilities to aid in the detection of inconsistencies to avoid missing design-specific consistency requirements.

If the Verifier finds no discrepancies, then we determine that the file system has recovered to a valid state; otherwise, we determine that a consistency bug has been found. We then perform further manual analysis to identify the root cause of such consistency bugs.

4.5 Fuzzing Loop

We organize the aforementioned components into a file system consistency fuzzing tool with an execution loop detailed as follows. For file system invocation sequence generation, we borrow components from popular kernel fuzzers, such as Syzkaller, including system call specifications and input executor programs. To generate and utilize on-disk states efficiently, we use a centralized file system image pool, and attach images to the running system on demand. This allows reduces the need for data transfers on invocation sequences during each execution cycle.

After generating the invocation sequence, we hook the virtual machine's block device drivers to intercept write operations to the target image. The invocation sequence is then executed, with an *umount()* call appended to trigger any remaining persistence operations. During this process, any write commands to the block device are intercepted and handled as described in the previous sections to produce test cases. Additionally, valid state generation and verification is performed as mentioned above. To encourage more diverse invocation sequence generation, we utilize kernel code coverage on only the affected file system modules as guidance, and preserve the invocations and the corresponding sequence for further generation attempts.

5 Implementation

We implemented SnapCC with borrowed components from Syzkaller, including system call specifications, seed storage, virtual machine management and execution feedback collection and processing. The core components of SnapCC, as described in the previous section, are written from scratch. The implementation details regarding SnapCC's design towards performing efficient crash state iteration are discussed in the following sections.

5.1 Copy-on-Write Based State Iteration

Thoroughly iterating through all possible states can be a performance challenge for SnapCC, for if the implementation used plain block operations to make a copy of all images produced, the runtime overhead would be too costly to perform dynamic testing. As aforementioned, we use snapshots that, instead of copying an entire file upon updates, require only that the modified pieces be recorded.

To facilitate efficient file system image snapshots for systematic state exploration for enumerating potential on-disk states and finding valid states to recover to, we employ Copy-on-Write mechanisms provided by QEMU's *qcow2* image format in conjunction with BTRFS's subvolume snapshot functionalities. Briefly put, using QEMU's *qcow2* format, which inserts a mapping in between the actual file system image and the virtual machine, allows for modifications to a block device to reside only on the *qcow2* file, without any modifications to the original image. BTRFS's subvolume snapshot functionalities allow fast access to previous snapshots of the same file. Combining the two techniques allows for efficient access to all possible states of file system images during testing for consistency bugs.

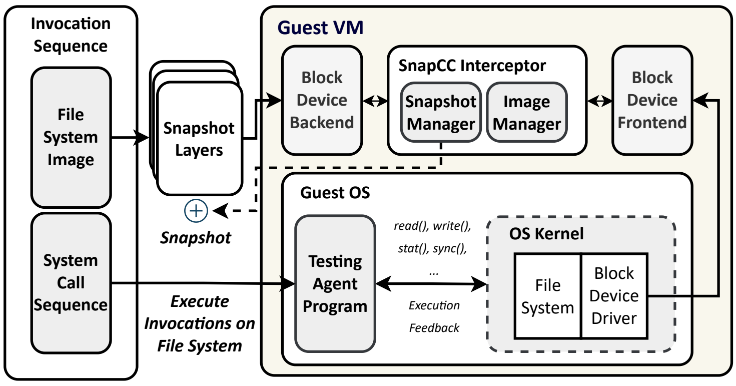


Fig. 5. SnapCC uses QEMU's qcow2 in conjunction with BTRFS snapshots to efficiently produce iterations of a file system invocation sequence.

The overall diagram is shown in Figure 5. Specifically, during state exploration for either identifying all possible crash points or finding all valid consistent states, we mount the initial file system image with a *qcow2* layer file on top to hold the modified data information and place it within a BTRFS subvolume. SnapCC then hooks QEMU's *qcow2* block device driver's write operations to perform snapshotting operations on demand. The block device is then attached to the virtual machine instance, which mounts it accordingly through the agent program. After preparations are complete, SnapCC initiates a test run, which instructs the agent program to invoke the system calls as described in the invocation sequence. For iterating through all possible crash points, each write operation signals SnapCC to trigger a snapshot invocation to record such states efficiently. For finding all valid consistent states, the agent program safely unmounting the image at the end of each persistent operation sequence triggers an immediate snapshot invocation from SnapCC.

After executing the invocation sequences, SnapCC can subsequently retrieve the desired file system images by layering the *qcow2* layer file stored in the corresponding subvolume snapshot and the base image.

5.2 Runtime Image Snapshot Caching

Producing and handling such an amount of snapshots during runtime takes its toll in runtime overheads. Appending synchronization operations after file system modifying operations alone constitutes a significant increase in runtime, due to the I/O operations involved with file system data persistence and snapshotting. This operation will need to run for each testing iteration, and thus will hamper the execution throughput, and, consequently, testing effectiveness.

To mitigate this issue, we use the following intuition: we can cache the relevant *qcow2* image files to avoid repeated computation on the same data. During fuzzing, many of the inputs executed are seeds from a previous run mutated in a few spots, such as an appended system call or modified arguments. Therefore, we observe that by caching the obtained file system images during previous executions, we can greatly reduce the computation required to execute a single invocation sequence.

Therefore, SnapCC utilizes image snapshot caching within its seed pool and utilizes the cached image states to reduce overheads. Specifically, when SnapCC generates a new invocation sequence from scratch, the valid state file system images and the possible bug point images are saved together with the seed, with each image pointing to the persistence operation that produced it. Utilizing the Copy-on-Write features introduced in the previous section, SnapCC's caching is extremely lightweight and runtime efficient, generally only having to process a fraction of the data of the original image. During execution, SnapCC is able to reuse snapshots starting from the first system call consecutively until the first mutated input. Furthermore, executing mutated inputs can be performed by starting with the immediate previous snapshot to the first mutated input and resuming execution from the next consecutive system call succeeding the persistence operation that produced the snapshot.

Therefore, a combination of Copy-on-Write snapshotting and image caching is in place to greatly relieve SnapCC of the potential overheads from the designed techniques.

6 Evaluation

To evaluate the effectiveness of our approach, we designed experiments to evaluate the bug finding capabilities and runtime performance of SnapCC pitched against the state-of-the-art consistency testers B3 and Hydra. As SnapCC's many components are borrowed from Syzkaller, we include Syzkaller in our experiments as a baseline subject. To thoroughly assess SnapCC's effectiveness, we propose the following research questions to guide our evaluations.

- **RQ1**: Does SnapCC explore more file-system-relevant code than the state-of-the-art?
- **RQ2**: Is SnapCC capable of finding consistency bugs in real-world scenarios?
- **RQ3**: How significant is the overhead of SnapCC by systematically enumerating potential on-disk states?

These research questions address the real-world performance of our approach, component-wise effectiveness and efficiency, as well as runtime optimization evaluation, which is sufficient to demonstrate the effectiveness of our approach.

6.1 Experiment Setup

We evaluate SnapCC's testing effectiveness against B3 (implemented as ACE and CrashMonkey) and Hydra. We also bring Syzkaller into our tests as a baseline. As SnapCC supports invoking *ioctl()* calls, which exceed the scope of B3 and Hydra, we also prepared SnapCC-, which is SnapCC with *ioctl()* generation and mutation capabilities disabled. To ensure fairness, we add the system call descriptions previously extended for SnapCC to Syzkaller. The machine used during the experiments is equipped with an AMD Ryzen 7 5800X processor, 32GiB of RAM, and running Arch Linux with kernel version 6.3. The kernel which hosted the file systems under test is Linux 6.2. We enabled the following file systems during kernel pre-compilation configuration: BTRFS, F2FS, and ext4. The kernel is also built with KASAN and KCOV enabled, for detection of memory address violations and retrieving coverage information. The coverage experiments were conducted over 24 hours, with data points sampled per each 30 minutes. Each real-world bug-finding experiment was conducted over the course of 14 days, totaling 336 hours per experiment. For experiments with B3, as it runs in a manner similar to batch programs rather than interactively, we executed B3 until its completion. Each real-world experiment instance contains five identical trials to reduce statistical errors. The results have been tested using the Mann-Whitney U Test to demonstrate statistical significance between different data groups, where any non-significant data points will be declared accordingly.

6.2 File System Module Code Coverage

We address **RQ1** by measuring code coverage for file system modules in the Linux kernel by SnapCC, in comparison to Hydra and B3, where the latter is executed after generating all possible

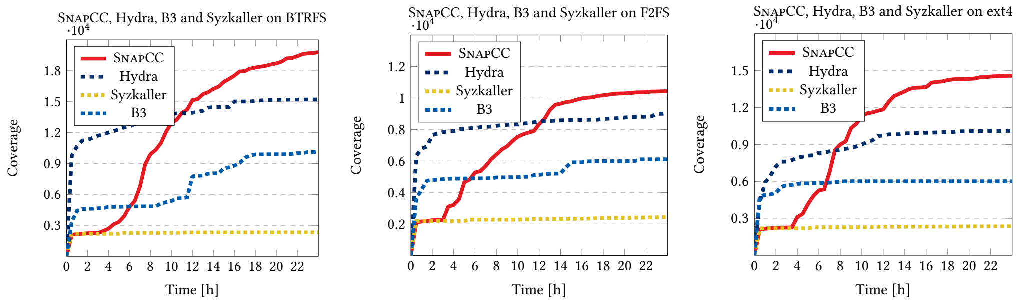


Fig. 6. Coverage of SNAPCC, Hydra, B3 and Syzkaller on Linux file systems BTRFS, F2FS and ext4 over 24 hours.

workloads, and Syzkaller, as a baseline fuzzer. For SNAPCC and Hydra, the coverage statistics are collected over the entire fuzzing campaign, whereas for B3, We collect coverage either up until 24 hours, as the other fuzzers are, or when it completes its execution. The coverage growth results are shown in Figure 6. As is evident from the graph, SNAPCC's coverage steadily grows to overtake B3's and Hydra's coverage statistics. Eventually, SNAPCC achieves a increase over Hydra of 30%, 16% and 44% improvement in BTRFS, F2FS and ext4, respectively. Syzkaller on the other hand, remains relatively constant, as it is not well-suited towards generating system call sequences and images that effectively explore the state space. We believe that SNAPCC's growth characteristics are due to the following reasons: first, SNAPCC, like Syzkaller, runs the target OS in a virtualized environment, whereas Hydra uses LibOS; second, the state space that SNAPCC can explore is broader due to the system call specifications provided, while Hydra mainly uses POSIX-compliant file system invocations. We will further discuss the relevant execution throughput statistics in the subsequent sections.

To address the concern that coverage improvement is the sole result of the inclusion of *ioctl()* system call specifications into the generation process, we perform an ablation test, one which compares SNAPCC-'s coverage on the same dataset to that of B3 and Hydra. thus giving the tools equal footing on restricted interaction with the target file system only through POSIX compliant system calls. SNAPCC-'s coverage statistics, along with those of Hydra and B3, are shown in Figure 7.

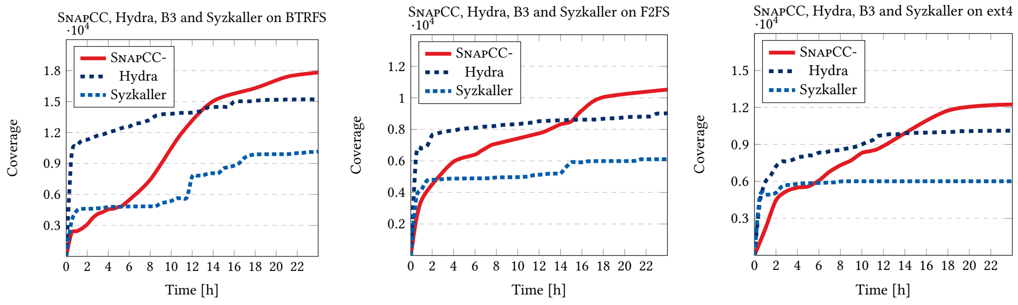


Fig. 7. Coverage of SNAPCC-, Hydra, B3 on Linux file systems BTRFS, F2FS and ext4 over 24 hours.

We can observe from the figure that while SNAPCC-'s coverage falls behind B3 and Hydra initially, it gradually overtakes them, as does the original version of SNAPCC. The difference is the amount of coverage improvement. Without *ioctl()* calls, we see that SNAPCC- overtakes B3 and Hydra, but at the end of 24 hours, yields improvements of 17%, 4%, and 21% compared to Hydra in BTRFS, F2FS, and ext4, shy of what the original version of SnapCC achieves. Our analysis of the its coverage shows that for the file systems, removing *ioctl()* impacts BTRFS the most, as many of its features that can only be reached by triggering the kernel function *btrfs_ioctl()*, which has been covered by SnapCC, are now unreachable. Regardless, we find that our systematic on-disk state exploration technique still allows SnapCC to cover more recovery and consistency checking code in the file system, and in turn have more chances in detecting bugs in its consistency checking and repairing code.

Therefore, SnapCC's better coverage statistics than the competition allows us to positively answer **RQ1**.

6.3 Consistency Bug Detection

To address **RQ2**, we first compile a dataset consisting of all historical crash consistency bugs found by B3 and Hydra, and test if SnapCC is able to find the relevant bugs. Then, we experiment with running SnapCC, SnapCC-, Hydra and B3 using real-world settings on Linux kernel version 6.2 to try uncovering consistency bugs within its file system implementations.

6.3.1 Historical Bug Dataset. We first collected 26 historical bugs found by B3 and 32 found by Hydra, constructed equivalent runtime environments and executed SnapCC for equivalent runtime environments to that of Hydra and B3. For the 26 historical bugs found by B3, SnapCC found 25, with the sole outlier being one requiring *dropcaches*, for which SnapCC currently lacks support. This result demonstrates that SnapCC is at least as effective in finding historical crash consistency bugs as B3 and Hydra.

6.3.2 Real-World Bug Detection Comparison. The statistics of the new consistency bugs, and in addition all new auxiliary bugs (including memory violations and kernel-defined bugs) found during the testing campaign are shown in Table 1. All bugs listed in the table have been responsibly reported according to common practice to the various kernel and file system maintainers for further analysis and patching.

As is evident in the table, SnapCC demonstrated excellent results in discovering new consistency bugs in real-world scenarios on recent versions of the Linux kernel, finding a total of 15 consistency bugs across widely-used file systems including BTRFS, F2FS, and ext4. In comparison, Hydra was able to detect 6 of the consistency bugs, while B3's performance is limited to only detecting 2 consistency bugs. SnapCC- found only 8 of the crash consistency bugs, based on our analysis, is mainly due to it not being able to trigger file-system-implementation-specific functionalities, and therefore cannot generate on-disk states as diverse as SnapCC, but is still better than Hydra and B3, demonstrating the effectiveness of systematically finding on-disk states as test cases to the file system. This clearly demonstrates that, as a whole, SnapCC's consistency bug detection capabilities are state-of-the-art.

The bugs found by SnapCC have all been confirmed by the relevant maintainers. Currently, 7 of them have been fixed, with the patches merged upstream to the Linux kernel, 6 have been confirmed, but due to reasons such as requiring a substantial re-work of their file system internals, are still under discussion of how the bugs should be fixed. The remaining 2 have been confirmed by the BTRFS maintainers, but they explained that these bugs won't require a fix, as the relevant modules are undergoing a complete rewrite and refactor.

To further assess the influence of using *ioctl()* system calls in SnapCC, we also tested SnapCC-'s bug finding effectiveness at the same time. Our results show that it only finds 4 of the 15 new bugs found by SnapCC's original version. Our analysis shows that the remaining bugs found by SnapCC requires the use of *ioctl()* to trigger a file-system-specific operation, such as rebalancing, snapshotting, or subvolume maintenance, or other *ioctl()*-enabled states, to enter an on-disk state that, when checked and fixed by the file system's checker, still produces a resulting file system image with inconsistencies.

Table 1. List of new bugs found by Syzkaller, SnapCC, SnapCC-, Hydra and B3 classified by bug type.

Bug Type	FS	Syzkaller	**SnapCC**	Hydra	B3	SnapCC-
Consistency	BTRFS	0	**9**	3	1	4
	ext4	0	**4**	2	1	2
	F2FS	0	**2**	1	0	2
Memory Error	BTRFS	1	**2**	4	0	2
	ext4	0	**2**	3	0	1
	F2FS	1	**1**	2	0	0
Kernel Bug	BTRFS	1	**2**	2	0	1
	ext4	1	**1**	0	0	1
Total Consistency		**0**	**15**	**6**	**2**	**8**

While finding memory violation bugs and kernel bugs (triggered using the `BUG_ON()` macro) are not the target bug type, as KASAN is enabled and kernel bugs appear when triggered, we nonetheless collect and display the new bugs found as well in the table. In this case, Hydra performs better than SnapCC, whereas B3 is incapable of triggering any such bugs. We believe the reason for this is due to Hydra, which tests multiple properties of file systems rather than consistency bugs alone, employs many techniques that would not be applicable to our scenario, such as image mutation. Interestingly, Syzkaller's effectiveness here is less pronounced than SnapCC's, mainly due to its design not being able to effectively trigger the multitude of crash states, thus it does not trigger bugs located within the file system checking and recovery routines well.

We further analyzed the characteristics of the newly found consistency bugs to identify probable reasons as to why the tools performed in this manner.

As B3 limits the number of file system invocations that it generates and performs the lowest of the three comparison tools, we first wish to establish an understanding regarding the distribution of the lengths of the new bug's reproducing system calls and how it may affect the performance of the tools. The relevant results, along with the number of new bugs of the specific lengths that each individual tool was able to trigger, are shown in Figure 8.

As shown in the graph, we observe that the majority of new bugs found require a substantial number of system calls in order to trigger, which is unfortunate for B3, as it can only generate length-bounded file system invocations, thus explaining its relatively unimpressive results. Hydra and SnapCC can generate longer system call sequences, as a result of employing fuzzing and thus explore an infinite input space.

However, we see that SnapCC triggers significantly more bugs through the use of 16 or more system calls than the comparison tools. Our preliminary analysis shows that these calls consist of significant numbers of *ioctl()* calls to invoke advanced functionalities in file systems, which is a task that Hydra performs less than optimal. To determine the root cause of this difference in performance, we analyzed the composition of specialized *ioctl()* calls in the bug reproducer programs of the newly found consistency bugs. The results are shown in Figure 9.

As shown in the graph, the newly discovered bugs require some usage of file system specific invocations, such as BTRFS in the chart, where some bugs already require manipulating snapshots or subvolumes, demonstrating the effectiveness of utilizing domain expert written specifications.

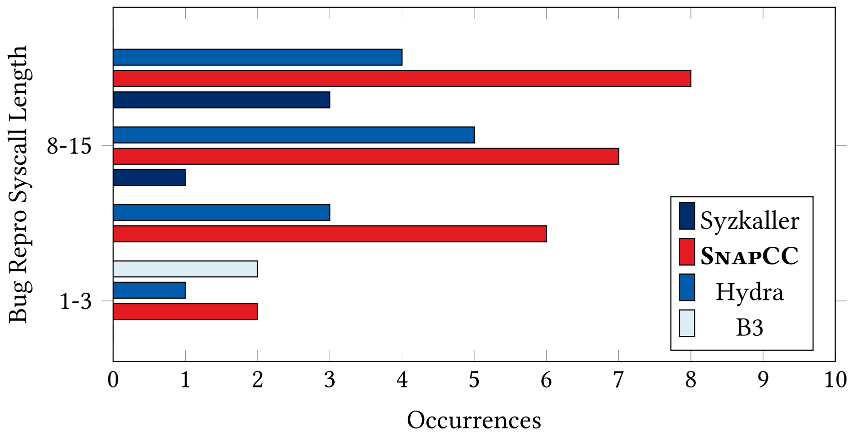


Fig. 8. Distribution of system call sequence length for the new bugs and the respective number found by SnapCC, B3 and Hydra.

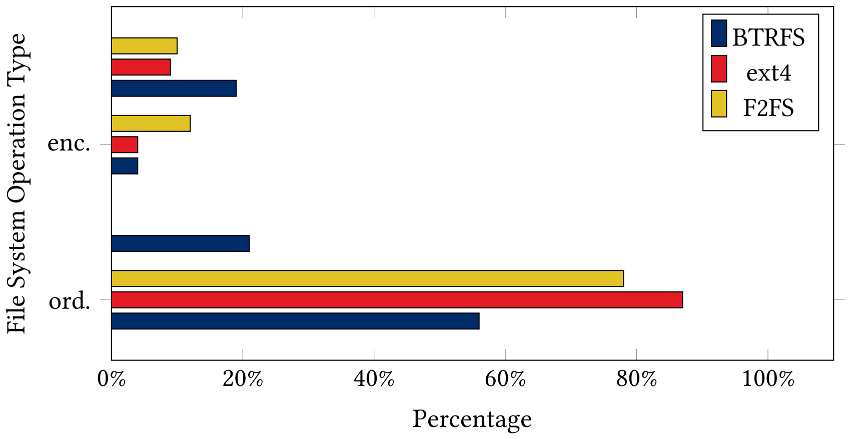


Fig. 9. Distribution of file system specific system calls in the bugs' reproducer programs per file system tested. ord. represents ordinary POSIX file system invocations; snap. referse to snapshot-relevant operations, enc. represents encryption/decryption related block operations, and vol. represent volume relevant operations.

However, there should be more influences towards the performance differences. To verify whether Hydra and B3 are capable of detecting the bugs that SnapCC uniquely found during this experiment, we let Hydra and B3's CrashMonkey run the reproducer programs of SnapCC's uniquely found bugs and attempt to trigger them through their own fault injection mechanisms. As their fault injection processes are randomized, we repeat the experiments for 50 times. The success rates are shown in Table 2. As demonstrated in the table, B3's CrashMonkey can find one such bug in BTRFS, while Hydra can find 2, where they are from BTRFS and ext4, respectively. However, their reproduction rates are relatively low, which is a result of randomly picking crashing points. In contrast, SnapCC traverses all possible on-disk states and verifies the states, allowing testers to concretely find consistency bugs.

Table 2. Bug reproduction rates of Hydra and B3 on SnapCC's uniquely found consistency bugs.

File System	SnapCC Unique Bugs Total	Hydra		B3	
		Once	Rate	Once	Rate
BTRFS	6	1	6%	1	2%
ext4	2	1	2%	0	0%
F2FS	1	0	0%	0	0%

6.3.3 Adaptability. Apart from the comparison file systems, we also wish to evaluate how SnapCC can adapt to other real-world file systems. Therefore, we added support for Linux's UFS, XFS, BCacheFS and OpenZFS file systems. The time required for us to support these file systems took us 15 minutes, 10 minutes, 1.5 hours, and 2 hours, respectively. The efficiency at which we adapted to these file systems demonstrate SnapCC's versatility, as most of its components are generalized across different file system implementations, and our use of file-system-specific checking tools have also removed the burden of customized consistency checking implementations.

We also conducted an experiment on these file systems to examine SnapCC's capabilities in finding consistency bugs. Our testing run of 72 hours yielded a total of 7 bugs in these file systems, specifically 1, 0, 5, 1, for UFS, XFS, BCacheFS, and OpenZFS, respectively. The bugs have also been reported and confirmed, where BCacheFS's bugs have all received confirmation that they will be fixed.

6.3.4 Consistency Bug Case Study. We use a crash consistency bug found in BCacheFS to demonstrate how SnapCC was able to detect file-system-implementation-specific crash consistency bugs. The bug itself is caused by the following file system invocations:

```
mkdir("d1"); mkdir("d2");
ioctl@BCH_IOCTL_SUBVOLUME_CREATE(“d1”);
fd1=open(“./d1/f1”); fd2=open(“./d2/f2”);
rename(“./d1”, “./d2/rd1”);
write(fd1, ...); write(fd2, ...); unlink(“./d2/rd1/f1”);
ioctl@BCH_IOCTL_SUBVOLUME_DESTROY(“./d2/rd1”);
sync();
```

During the synchronization process, when SnapCC intercepts a persistence operation to obtain an on-disk image and mounts the file system in Linux, the file system encounters a lingering inode problem, where ./d2/rd1/f1's inode persists while the subvolume is destroyed when it should have been removed with the last three operations. This is due to a bug in BCacheFS's *fsck_write_inode()* function, which erroneously writes the inode back into the fixed file system. SnapCC found this bug by generating an invocations sequence containing ioctl() calls that trigger BCacheFS's snapshot creation and destruction operations, systematically finding all possible on-disk states and finding the correct file system image, and automatically determining valid states based on this sequence for validation. The bug has been fixed in patches for Linux's 6.12-rc2.

Both B3 and Hydra cannot find this bug, as they cannot generate invocations to *ioctl()*, which invoke subvolume creation and deletion. Even if they have been retrofitted with *ioctl()* generation capabilities, it is difficult to discover this bug due to the amount of operations committed during this process for fault injection, and the valid states that need to be detected automatically for detecting this bug.

6.3.5 Summary. SnapCC's improvement is the contribution of many factors combined, including a more effective input generation and mutation mechanism, as well as systematically enumerating all possible on-disk states during an invocation sequence's execution. Therefore, we have answered **RQ2** that SnapCC is indeed capable of finding consistency bugs in real-world scenarios with state-of-the-art effectiveness.

6.4 Systematic On-Disk State Explorer Overhead

To assess the impact of systematically enumerating each possible on-disk state during SnapCC's execution process, we compare the execution time of SnapCC's On-Disk States Explorer against Hydra's comparable fault injection mechanisms. Instead of executing the entire fuzzing loop, we take the seeds cumulated over SnapCC's real-world experiments and run them separately. During their executions, we measure the elapsed time for SnapCC and Hydra to finish finding possible on-disk states or conducting fault injections. The data is grouped according to the length of the invocation sequence. The average results are shown in Table 3.

Table 3. Overhead Comparison Between SnapCC and Hydra's fault injection process.

Fuzzer	Elapsed Time per Sequence (ms)			
	1-3	4-7	8-15	16+
SnapCC	52.3	88.7	102.5	137.6
Hydra	27.6	47.2	68.5	91.2

The results show that while SnapCC exhibits a higher overall overhead, SnapCC compensates by being more effective in rooting our consistency bugs during each execution iteration, which is evident in the statistics found in the previous section. Therefore, we can answer **RQ2** that while SnapCC's overhead is higher than Hydra's, SnapCC delivers more testing effectiveness per execution through systematic traversal of possible on-disk states.

7 Discussion

7.1 Manual efforts involved

We have greatly reduced the manual labor involved in testing file system consistency bugs with SnapCC. Specifically, the only places requiring such labor is listed below. When one wishes to test file systems for consistency errors, the manual process they need to go through is to prepare kernel compilation parameters to include relevant modules and debug options. Further possible human intervention is writing new system call specifications for newly added file systems. Other operations, such as initiating the test process, result processing and bug reporting, like other state-of-the-art tools, can be initiated either manually or through pre-written scripts, thus alleviating the burden to an extent.

7.2 File system image manipulation

Intuitively, during input mutation, mutating the file system image along with the system call sequence may yield more bugs, as the input space can be explored more easily. Currently, our approach does not manipulate the initial file system images randomly, but rather relies on resulting images from system call executions as new images for further execution. Our belief is that consistency needs to be guaranteed initially for the test to be valid and bugs be sound. However, this can potentially lower our performance, whereas in contrast, Hydra manipulates the image directly in conjunction with generating invocation sequences. Further work for SnapCC may revolve around designing efficient image manipulation methods to accelerate the fuzzing process.

7.3 Soundness and Completeness

There exists the possibility of initially detecting false positives or false negatives in crash consistency bugs, due to there existing potential interfering threads in the system that cause noise in the file system, such as concurrent processes that perform indexing, etc. These events result in false positives and negatives, as they affect the persistence operations of the file system, and produce different on-disk states and valid states for testing and validation. SnapCC has mechanisms to mitigate this issue, which performs a deflaking process, where it re-runs the entire process of finding valid on-disk states and valid file system states repeatedly for a fixed number of times (usually 5). This process itself under most circumstances can eliminate the effects of such events and remove false positives and negatives.

Additionally, journaling and transactional file systems may have intermediate states that differ from a sequential file operation view. Our approach is based on the abstractions afforded to us by the file system interfaces, and we do not make any assumptions on the valid state beneath the abstractions and into the file-system-specific implementations. If the file system has implementation-specific logic that results in states differing from other file systems, the set of valid states will contain this state for crash consistency verification, as it is present after a write operation combined with a persistence operation, and thus captured by SnapCC's mechanisms.

8 Related Work

8.1 Fuzzing

General purpose fuzzing has the attracted the most attention from either fields due to it being the most applicable technique to most types of software testing, while new techniques such as mutation operators, scheduling algorithms and feedback indicators can be tested and refined.

One of the most widely-used general-purpose fuzzers is AFL [17], which pioneered many technical ideas that have affected the design of fuzzers, such as edge-based coverage feedback, fork-server-based rapid testing, etc. AFL++ [9] is a popular state-of-the-art fuzzer based on AFL that has incorporated many ideas and techniques from academia, such as REDQUEEN [1], AFLFast [2], etc.

libFuzzer [26] is another popular fuzzer that is also very effective in testing a wide variety of software applications. In contrast to AFL, libFuzzer requires the program-under-test to provide a *LLVMFuzzerTestOneInput()* interface, which allows libFuzzer to input test data into the program-under-test, thus rendering its fuzzing effectiveness highly dependent on the quality and scope of the fuzzing interface functions. FuzzGen [13] and IntelliGen [35] have been proposed to automatically or semi-automatically synthesize fuzzing driver programs, i.e. programs that take inputs generated from fuzzers and direct them to relevant library functions, to assist with fuzzing such targets.

There is also a branch of fuzzing that utilizes hybrid execution with symbolic execution and SAT/SMT solvers to increase code coverage and find more difficult-to-trigger bugs. One prominent example is QSYM [34], which utilizes a concolic executor to help fuzzers through predicates they have difficulty solving. Another example is Angora [6], which uses dynamic taint analysis coupled with a gradient descent solver to increase code coverage.

Many works also focus on scheduling seed between different fuzzers as a means to enforce cooperation between different fuzzers, such as EnFuzz [7]. Seed scheduling also shows promise in increasing fuzzing performance, such as Mopt [19], which has been integrated into AFL++.

8.2 Kernel & File System Testing

Syzkaller is a kernel fuzzer that has been the inspiration for many. HEALER [31] is a Syzkaller-inspired kernel fuzzer, where it learns the relations between system calls to produce inputs with higher quality. Moonshine [25] is another fuzzer based on Syzkaller that distills a program's system call traces to obtain a high quality inital seed. Other types of kernel fuzzers include kAFL [28], a kernel fuzzer that utilizes hardware features to accelerate fuzzing, TriforceAFL [12], which augments AFL with full-system fuzzing capabilities, etc. Embedded kernels have also attracted the attention of fuzzing testers, including Gustave [8] and Tardis [29].

File system testing mainly involves testing for three classes of bugs: memory errors, semantic bugs, and consistency bugs. Kernel fuzzers such as Syzkaller and file system fuzzers such as Hydra can detect memory errors efficiently using tools such as KASAN [11]. Semantic bugs can be found either through static analysis [21], dynamic testing, or formal verification [24, 30, 33]. Apart from these, *xfstests* [20] is a unit test suite that examines file systems for common errors. Generating workloads is crucially important for uncovering potential bugs, such as Chen et al.'s work [5]. Also, work surrounding formal specifications and verifications in consistency checking is also active, such as Bornholt et al's work [3]. File system consistency checkers are also actively researched, such as Gatla et al.'s [10] and Menezes Carreira et al.'s work [4]. Persistent memory crash consistency is also important, such as Chipmunk [18].

8.3 Consistency Checking

Jiang et al.'s[14] main focus is developing automated consistency oracles for user space applications. While in principle, the approach may be similar conceptually, file systems require special attention to operating system-level abstractions, such as inode consistency. Therefore, we use a block-level mechanism to capture all possible persistent states, and use inode-level comparisons on file contents, metadata, and directory structure to verify the consistency.

9 Conclusion

In this paper, we present SnapCC, a new approach towards effective file system consistency testing. In contrast to previous works, SnapCC proposes the use of systematic state exploration to further uncover consistency errors within modern file systems. Through the design and implementation of a system-call-based File System Invocation Sequence Generator, a Systematic On-Disk State Explorer, and an Automatic Valid State Finder, SnapCC systematically finds all possible on-disk states during the execution of a file system invocation sequence and automatically determines all valid file system states. Through comparing the entries of the two sets and identifying any condition that is not valid, SnapCC effectively finds consistency bugs in file systems. SnapCC's capabilities are demonstrated through our evaluations, where it shows a 16% to 44% increase in coverage statistics over Hydra, in addition to the 15 new crash-consistency bugs found in widely-used file systems BTRFS, F2FS, and ext4. We also found 7 new crash consistency bugs in other Linux file systems UFS, XFS, BCacheFS, and OpenZFS, showing SnapCC's adaptability to file systems. Additionally, we established that SnapCC's effectiveness is due to both invocation sequence generation and systematic on-disk state exploration, being able to trigger more states in the file system's code.

10 Acknowledgments

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), and NSFC Program (No. U2441238, 62021002).

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS).*

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16).* Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[3] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16).* Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/2872362.2872406

[4] João Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. 2012. Scalable Testing of File System Checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12).* Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2168836.2168861

[5] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Testing File System Implementations on Layered Models. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20).* Association for Computing Machinery, New York, NY, USA, 1483–1495. https://doi.org/10.1145/3377811.3380350

[6] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP).* 711–725. https://doi.org/10.1109/SP.2018.00046

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19).* USENIX Association, USA, 1967–1983.

[8] Stéphane Duverger and Anaïs Gantet. 2021. GUSTAVE: Fuzz It Like It's App. *DMU Cyber Week* (2021).

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies.* 10–10.

[10] Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. 2018. Towards Robust File System Checkers. *ACM Trans. Storage* 14, 4, Article 35 (dec 2018), 25 pages. https://doi.org/10.1145/3281031

[11] Google. [n. d.]. Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[12] Jesse Hertz and Tim Newsham. 2019. TriforceAFL. AFL Qemu fuzzing with full-system emulation. *QEMU Fuzzing With Full-System Emulation* (2019).

[13] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20).* 2271–2287.

[14] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. Crash consistency validation made easy. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016).* Association for Computing Machinery, New York, NY, USA, 133–143. https://doi.org/10.1145/2950290.2950327

[15] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19).* Association for Computing Machinery, New York, NY, USA, 147–161. https://doi.org/10.1145/3341301.3359662

[16] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2020. Finding Bugs in File Systems with an Extensible Fuzzing Framework. *ACM Trans. Storage* 16, 2, Article 10 (may 2020), 35 pages. https://doi.org/10.1145/3391202

[17] lcamtuf. 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[18] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23).* Association for Computing Machinery, New York, NY, USA, 718–733. https://doi.org/10.1145/3552326.3567498

[19] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19).* USENIX Association, USA, 1949–1966.

[20] Linux Kernel Maintainers. [n. d.]. xfstests. https://github.com/kdave/xfstests.

[21] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15).* Association for Computing Machinery, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422

[22] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 33–50.

[23] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2019. CrashMonkey and ACE: Systematically Testing File-System Crash Consistency. *ACM Trans. Storage* 15, 2, Article 14 (apr 2019), 34 pages. https://doi.org/10.1145/3320275

[24] Jan Tobias Mühlberg and Gerald Lüttgen. 2012. Verifying Compiled File System Code. *Form. Asp. Comput.* 24, 3 (may 2012), 375–391. https://doi.org/10.1007/s00165-011-0198-z

[25] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[26] The LLVM Project. [n. d.]. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[27] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3, Article 9 (aug 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[28] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[29] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. 2022. Tardis: Coverage-Guided Embedded Operating System Fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), 1–1. https://doi.org/10.1109/TCAD.2022.3198910

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

[31] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[32] Dmitry Vyukov and Andrey Konovalov. 2015. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[33] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 393–423.

[34] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[35] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327.