Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework

Seulbae Kim Meng Xu Sanidhya Kashyap Jungyeon Yoon Wen Xu Taesoo Kim
Georgia Institute of Technology

Abstract

File systems are too large to be bug free. Although handwritten test suites have been widely used to stress file systems, they can hardly keep up with the rapid increase in file system size and complexity, leading to new bugs being introduced and reported regularly. These bugs come in various flavors: simple buffer overflows to sophisticated semantic bugs. Although bug-specific checkers exist, they generally lack a way to explore file system states thoroughly. More importantly, no turnkey solution exists that unifies the checking effort of various aspects of a file system under one umbrella.

In this paper, we highlight the potential of applying fuzzing to find not just memory errors but, in theory, any type of file system bugs with an extensible fuzzing framework: Hydra. Hydra provides building blocks for file system fuzzing, including input mutators, feedback engines, a libOS-based executor, and a bug reproducer with test case minimization. As a result, developers only need to focus on building the core logic for finding bugs of their own interests. We showcase the effectiveness of Hydra with four checkers that hunt crash inconsistency, POSIX violations, logic assertion failures, and memory errors. So far, Hydra has discovered 91 new bugs in Linux file systems, including one in a verified file system (FSCQ), as well as four POSIX violations.

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

Keywords File systems; semantic bugs; fuzzing;

ACM Reference Format:
Seulbae Kim Meng Xu Sanidhya Kashyap Jungyeon Yoon Wen Xu Taesoo Kim . 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359662

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6873-5/19/10...$15.00
https://doi.org/10.1145/3341301.3359662

1 Introduction

Designing and maintaining file systems are complicated. With the constant development for performance optimizations and new features, popular file systems have grown too large to be bug-free. For example, `ext4` [6] and `Btrfs` [44], with 50K and 130K lines of code, respectively, witnessed 54 [26] and 113 [25] bugs reported in 2018 alone. A bug in a file system can wreak havoc on the user, as it not only results in reboots, deadlock, or corruption of the whole system [30], but also poses severe security threats [32, 51, 53]. Thus, finding and fixing bugs is a constant yet essential activity during the entire life cycle of any file system.

However, manually eliminating every bug in file systems with such massive codebases is challenging. For most file systems, the best effort in getting assurance that no obvious bugs are introduced is to rely on known regression tests (*e.g.*, `xfstests` [42]), and tools (`fsck` [50]). However, these tools cannot handle the diverse types of semantic bugs applicable to file systems. For example, we found a case in `Btrfs` (Figure 1) that could cause irrecoverable data loss in the event of power loss or system crashes, which can be disastrous for data centers. Both `xfstests` and `fsck` miss this case. In fact, only 5% of tests in `xfstests` attempt to stress such scenarios, which is not sufficient. While specialized checkers can often complement manually written cases in capturing semantic bugs [33, 43], they face the common problem of generating test cases that thoroughly explore the file system codebase. More importantly, there is no turnkey solution that packs different checkers seamlessly and fits into the continuous integration process of file systems.

Recently, the decades-old software testing technique—fuzzing [3, 41, 57]—has become the go-to approach, with thousands of vulnerabilities in real-world software, including critical ones [56], as its trophies. Without a doubt, file systems can be fuzzed, and existing OS fuzzers [17, 23, 35, 46] have demonstrated this viability with more than 50 bugs found. Our recent file system-specific fuzzer further boosts the bug count by 90 [51]. However, all prior works on file system fuzzing have focused on memory safety bugs only, forgoing the opportunity to find the dominant, diverse, and harder-to-catch category of file system bugs: semantic bugs [30].

Semantic bugs in file systems come in various flavors, including but not limited to violations of widely agreed properties (*e.g.*, crash-safety), non-conformance to specifications (*e.g.*, POSIX standard), and incorrect assumptions made by

developers. Therefore, different types of semantic bugs often require specialized checkers to find them. However, one shared feature among semantic bugs is that when triggered, they are unlikely to cause a kernel panic or hang, *i.e.*, no visible effects, at least in the short term. This contradicts with memory errors (*e.g.*, buffer overflow), which often lead to an immediate kernel panic once triggered. In fact, the property of silent failure hinders semantic bugs from being discovered by existing memory safety-oriented fuzzers.

To bridge the gap between fuzzing and semantic bugs in file systems, we build HYDRA, an extensible fuzzing framework that is capable of discovering any type of semantic bug (in theory) in various file systems with full automation. As a framework, HYDRA provides building blocks for file system fuzzing, including input mutators, feedback engines, a libOS-based executor, and a bug reproducer with test case minimization. HYDRA gains the capability of checking specific types of file system bugs by plugging in specialized checkers, which can be independently developed and integrated in different forms, such as an out-of-band emulator (*e.g.*, SibylFS oracle [43]) or inlined reference monitors (*e.g.*, `Btrfs` extent tree reference verifier [1]). In this way, bug checker developers may now focus on the core logic for hunting bugs of their own interest, while offloading file system state exploration and bug processing to HYDRA.

In this paper, we demonstrate how HYDRA goes beyond low-hanging memory errors to find three common types of semantic bugs in file systems: crash inconsistency, POSIX violations, and file system-specific logic bugs. These bugs are found by plugging into HYDRA an in-house crash consistency checker (see, §3.4), the SibylFS oracle [43], and the existing file system-specific assertions inside the codebase. As a result, HYDRA found eight crash inconsistencies, four POSIX violations, and 23 logic bugs in three popular and heavily tested Linux file systems (`ext4`, `Btrfs`, and `F2FS`) and even one crash consistency bug in FSCQ, which has been proven to not have such bugs [8]. HYDRA also finds 59 memory errors, while fuzzing several file systems.

Summary. This paper makes the following contributions:

- To tackle the dominant and diverse set of semantic bugs in file systems, we propose to use fuzzing as a one-stop solution that unifies existing and future bug checkers under one umbrella.
- To show this, we build HYDRA[1], a generic and extensible file system fuzzing framework that provides the supporting services for file system bug hunting so that developers can focus on writing core logic in checking bugs of their own interests.
- Leveraging in-house developed and externally available bug checkers, HYDRA has discovered 91 new bugs of four different types in various file systems, out of which 60 bugs have been acknowledged and 46 bugs have been fixed, which shows its worth.

[1]HYDRA is open-sourced at https://github.com/sslab-gatech/hydra

Bug	Description (w/ bug checkers)
CI	Data not properly persisted upon system crash or power loss **Bug checker:** SYMC3 (§3.4)★, eXplode [52], B3 [33]
SV	Implementation not conforming to specifications (e.g., POSIX) **Bug checker:** SibylFS [43]★, EnvyFS [19], Recon [11]
LB	Using wrong algorithms or making invalid assumptions **Bug checker:** Built-in checks (*e.g.*, [1])★
ME	Out-of-bound accesses, use-after-free, uninitialized read, etc **Bug checker:** KASan [13]★, KMSan [14], UBSan [45]

Table 1. A list of common types of file system bugs, their root causes, the corresponding checkers that can be plugged into HYDRA to find these bugs. ★ indicates bug checkers HYDRA integrated with.

```
1 mkdir("./A");
2 sync();
3 fd = open("./A/x", O_CREAT | O_RDWR, 0666);
4 pwrite64(fd, buf, 4000, 4000); // size: 8000
5 fdatasync(fd);
6 ftruncate(fd, 3000); // shrink size to 3000
7 rename("./A/x", "./y");
8 fsync(fd); // persist metadata (size: 3000)
```

(Crashing right after 8, the expected size of y was 3000, but found 8000)

Figure 1. A crash inconsistency bug in Btrfs that HYDRA found: `fsync()` fails to persist the size of a renamed inode. Data is corrupted as a consequence.

2 Background

File systems are complex and ever-growing artifacts. A recent study reveals that about 40% of patches in file system development are fixes for bugs of various kinds, reflecting both the diversity and severity of bugs in file systems [30]. In this section, we briefly explain four common types of file system bugs, introduce state-of-the-art bug-finding and elimination tools, and explain why fuzzing can be a turnkey solution to all types of bugs by complementing existing tools.

2.1 A Broad Spectrum of File System Bugs

Table 1 summarizes four types of bugs that are often found in mainstream file systems as well as related bug checkers that are specially designed for each bug type.

Crash inconsistency (CI). A file system is *crash consistent* if it always recovers to a correct state after a system crash due to a power loss or a kernel panic. A correct state means that the internal data structures are consistent and information that was explicitly persisted before the crash is neither lost nor corrupted. As a counter-example, Figure 1, reported by HYDRA, is a case that violates the crash consistency property because the size of the renamed inode is not persisted even after the completion of the explicit `fsync` call. Such types of bugs lead to devastating consequences: loss or corruption of persistent data and unmountable file systems, as well as duplicate/unremovable files. Therefore, crash consistency is a fundamental property relied upon by data-sensitive applications, such as databases or servers. Unfortunately, there are very limited testing resources for this

```
1 mkdir("./A", 511);
2 unlink("./A"); // fails to unlink
```

(expected to get EPERM in POSIX, but got EISDIR)

Figure 2. Specification non-conformance in ext4: `unlink()` returns an error code that is not compliant with POSIX, but it is acceptable to the Linux specification [43].

```
1 char buf0[8192] = { 0, };
2 fd = open("A/B/x", O_CREAT | O_RDWR, 0666);
3 fsync(fd);
4 symlink("A/B/acl", "./z");
5 fallocate(fd, 1, 6588, 7065);
6 write(fd, buf0, 2325);
7 fdatasync(fd);
8 link("A/B", "A/B/C/y");
9 rename("A/B/x", "A/B/C/y");
```

(failed to verify the extent tree refs)

Figure 3. Logic bug in Btrfs: a crafted image with a combination of syscalls results in a corrupted file extent tree.

property apart from a handful of regression tests and the recent work: eXplode [52], B3 [33].

Specification violation (SV). Specifications, such as POSIX standards or Linux man pages, are bridges between file system developers and users. Thus, a robust program must abide by the agreed-on specifications, which essentially are confined to what is allowed and not allowed out of a file operation. Figure 2 is a POSIX violation reported by Hydra, as the allowed error code is `EPERM`, while the actual implementation returns `EISDIR`. Note that this does not violate the Linux specifications. As in the example, if the file system does not conform with the specifications, the robustness and security of the software built and run on top of it can be critically affected (*e.g.*, by improper error handling). Nonetheless, similar to the crash inconsistency case, there are limited testing tools for specification violation checking apart from regression tests and the recent work, SibylFS [43].

Logic bug (LB). Unlike the other three bug types that can be defined independently of any specific file system, logic bugs are tightly coupled with the specific file system implementation. For example, the `F2FS` implementation requires its own notion of *rb-tree consistency* [55], which is not commonly asserted by other file systems. In other words, no pattern, such as an inconsistent state, deviation from POSIX standard, or simply crashes or hangs, exists to define a logic bug. However, similar to crash inconsistencies and POSIX violations, most logic bugs simply fail silently. Hydra found the case shown in Figure 3, which executes seemingly fine if the corresponding `Btrfs` extent check [1] is not enabled. However, such logic bugs not only lead to undefined behavior but also affect performance and reliability in the long run. File system developers are often aware of potential logic bugs and have placed extensive runtime assertions (*e.g.*, invariant checks) in the codebase to catch them. Unfortunately, such expensive checks are never enabled in production while existing file system test suites can rarely explore these corner states.

Memory errors (ME). Memory errors are common in file systems. Due to their high security impact, *e.g.*, enabling

```
1 chmod("A/B/x", 3072);
2 unlink("A/B/hln");  // hln hardlinks to file x (image setup)
3 open("A/y", O_CREAT | O_RDWR, 0666);
4 rename("A/y", "A/B/x");
```

(a use-after-free error caught by KASan)

Figure 4. Memory error in `ext4`: `chmod` brings `x` to `dcache`; `unlink` drops its `i_nlink` to 0 and moves it to the orphan list; `rename` frees the `inode` but its pointer is still in the orphan list; when unmounting, a use-after-free is detected.

remote code execution, several runtime checkers have been proposed to detect memory errors. The most prominent examples are the sanitizer series, *i.e.*, KASan [13], KMSan [14], and UBSan [45], to address out-of-bound accesses and use-after-free, uninitialized read, and undefined behaviors, respectively. Despite the scrutiny from sanitizers coupled with OS fuzzers, Hydra still finds a significant number of memory errors. Figure 4 illustrates just an example of triggering a use-after-free in the heavily checked `ext4` file system with a crafted image and as little as four syscalls.

Other types of bugs. File systems encounter even more types of bugs. For example, one major category is concurrency bugs, such as sleeping in atomic context, data races, deadlocks, and double-unlocks. Concurrency bugs have attracted a fair amount of attention from both industry and the research community. To discover such bugs, several dynamic checkers have been recently proposed, ranging from kernel built-in support (*e.g.*, `LOCKDEP` [34]), to industrial tools (*e.g.*, KTSan [16]), to research prototypes (*e.g.*, SKI [10]). Although our current demonstration focuses on the other four bug types (Table 1), in theory, Hydra can detect such bugs with appropriate checkers, which we leave as future work. Some file system bugs stem from disk-level failures [2, 9]. Although these bugs can be systematically detected [39, 52, 54], this is beyond the scope of this paper, as we assume that persistent storage is reliable.

2.2 Toward Taming Bugs in File Systems

Past years have witnessed numerous efforts in hardening file systems, ranging from comprehensive regression testing to bug-specific checkers and formal verification. Unfortunately, none of them have solved the problem entirely.

Regression tests. As the state of the practice, file system developers often rely on regression tests (*e.g.*, `ltp` [47] and `xfstests` [49]) and testing tools (*e.g.*, `fsck` [50]) to gain assurance of their implementation. Although this practice keeps growing, these test suites are still ad-hoc collections of tests that mostly focus on regression instead of systematic checking for file system semantics such as crash consistency, POSIX conformance, or file system-specific invariants. Moreover, handwritten test cases are far from sufficient to cover huge input space in file system execution. In fact, all the bugs Hydra found are missed in all of the test suites.

Bug-specific checkers. Recently developed bug-specific checkers have been successful in tackling hard-to-catch semantic bugs, such as B3 [33] in finding crash inconsistencies

```
1  mkdir("./A")
2  fd_foo = open("./A/foo", O_CREAT | O_RDWR, 0666);
3  link("./A/foo", "./A/foo_lnk");
4  sync();
5  fd_root = open(".", O_DIRECTORY, 0);
6  rename("./A/foo", "./y");
7  fsync(fd_root);
8  fd_x = open("./x", O_CREAT | O_RDWR, 0666);
9  fsync(fd_root);
10 pwrite64(fd_x, "aabbccdd", 8, 500); // size: 508
11 ftruncate(fd_x, 300); // shrink size to 300
12 unlink("./A/foo_lnk");
13 fd_y = open("./y", O_RDWR, 0);
14 fdatasync(fd_y);
15 fsync(fd_x); // persist metadata (size: 300)
```

Figure 5. `Btrfs`- `fsync` fails to persist the size of a truncated inode, in the presence of metadata changes for another inode. After a crash, `Btrfs` recovers file `x` to size 508, even though its truncated size, 300, should have been persisted.

and SibylFS [43] in finding POSIX violations. In fact, their effectiveness can be further boosted with a more efficient test case generator, as shown in the example in Figure 5.

In Figure 5, all 15 operations collectively trigger a crash consistency bug, *i.e.*, even though files `x` and `y` have no explicit correlation to each other, `fdatasync` on `y` (line 14) causes the metadata of `x` to be lost. B3 missed this bug due to an assumption in their workload generator: "maximum number of core operations in a workload is three," which is established under the observation that most crash consistency bugs are triggered by three or fewer core file operations. B3 enumerates all test cases that have one (seq-1) or two (seq-2) core operations, and a subset of seq-3 workloads. However, in order for B3's input generator to reach this bug, the bound needs to be lifted to generate seq-5 test cases. This not only contradicts B3's design choice to reduce the space of possible workloads, but also is infeasible, requiring a considerable amount of time, *i.e.*, more than a week if optimistic (§5.6).

Formal verification. Formally verified file systems have been promising candidates to put the bug-hunting war to an end given their attractive nature of being hassle-free by proof. Prominent examples include the FSCQ-family [7, 8, 21] and Yggdrasil [48] with different guarantees proved. Although we did not initially expect Hydra to find bugs in verified file systems, to our surprise, Hydra found two bugs in FSCQ (one of them is reported by B3 [33] as well), which cause the crash consistency property to be violated. There have also been works regarding formally verifying user-level software running on top of a file system [28], complementing the efforts of the formally verified file systems.

Motivations. Surveying existing approaches on file system bug finding reveals a common theme: the need for an efficient and practical explorer that traverses file system states in both breadth and depth, especially to reach corner cases that cannot be covered by testcases contemplated by human. With such an explorer, we could (1) harvest extensive invariant checks in the codebase to detect file system-specific logic bugs; (2) improve and complement existing bug detectors, *e.g.*, SibylFS; and more importantly, (3) focus on the core bug hunting logic and totally decouple state exploration, as shown by the improvements of our in-house crash consistency checker, SymC3, over B3 (§5.6).

2.3 Fuzzing as a Turnkey Solution

Fuzzing is a software-testing method that repeatedly generates new inputs for target program to trigger bugs. It is one of the most effective approaches in finding security bugs in software [3, 4, 12, 38, 57]. Moreover, there are several kernel-based fuzzers [17, 35, 46] that use mutated syscalls for fuzzing. What makes fuzzing unique over other bug-finding tools is its capability to generate interesting test cases with little domain knowledge. Fuzzy input mutators, inspired by genetic programming, are especially good at producing test cases that explore corner cases in program execution paths, which are otherwise difficult for humans to even contemplate. The execution feedback further directs the fuzzing effort toward both unexplored code paths and checker-desired states. Both properties are valued by bug checkers, as they boost the quality of test cases, which directly correlates to the number of bugs that can be found. Furthermore, given that test case generation often is not dependent on the core bug checking logic, it can be a perfect target to be offloaded to fuzzing.

This inspires us to design an extensible fuzzing framework that complements existing bug checkers with a fuzzing-based file system states explorer. Further experience in fuzzing OS kernels has shown unique challenges, such as reproducing bugs and creating proof-of-concepts (PoC) of reasonable size. Traditional OS fuzzers use virtualized instances to test file system functionalities. However, to avoid the expensive cost of rebooting a VM or reverting a snapshot, they re-use an OS or file system instance across multiple runs, causing the bugs found to carry the impact of thousands of syscalls and often become irreproducible. We would like to address these concerns in the framework as well.

3 Design

We now propose to build a comprehensive framework, Hydra, to complement existing and future bug checkers by providing a set of commonly required components, all tailored to the file system fuzzing. With proper checker plugins installed, Hydra is capable of stressing various aspects of a file system.

3.1 Hydra Overview

Figure 6 shows the components and workflow of Hydra. Hydra initiates fuzzing by selecting a seed from the seed pool. A seed consists of a file system image and a sequence of syscalls. The input mutator subsequently mutates either the image or the syscalls or both and produces a batch of test cases (§3.2). The test cases are sent to a library OS-based executor that always starts in a clean-slate state, mounts the given image, and executes the syscalls (§3.3). The visited code paths are profiled into a bitmap by the coverage tracker

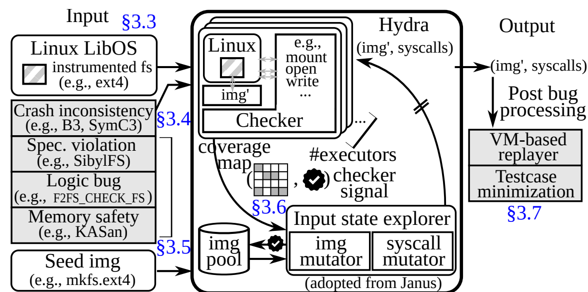

Figure 6. Overview of HYDRA's architecture and workflow.

instrumented when compiling the target file system. Meanwhile, the bug-checking dispatcher invokes the necessary checks, such as runtime assertions or an out-of-band emulation (§3.4, §3.5). The dispatcher later collects the checker's feedback and merges with the coverage bitmap into a fuzzing feedback report. If new coverage is reported or the checker marks the test case as interesting, the test case is saved in the seed pool and more exploration along this direction is expected; otherwise, the test case is discarded (§3.6). If a new bug is reported, the test case will be sent to a virtual machine for replay and confirmation. HYDRA also performs syscall sequence minimization on the test case to create a simplified PoC for the ease of analyzing and fixing the bug (§3.7).

3.2 Input Mutator

To trigger a file system bug, one needs to (1) mount a corrupted image or (2) execute a crafted sequence of syscalls, or most likely, (3) combine both steps. To exploit the synergy between mutating both file system images and syscall traces, we incorporated our previous work, Janus [51], into HYDRA.

Image mutation. A file system image is highly structured into user data chunks (e.g., file content) and a few management structures called *metadata*. For mounting and a majority of file operations, only metadata is consumed, which constitutes merely 1% of the image size, making a feasible target for random mutation. Using these facts, HYDRA scans the image to locate all metadata blobs using a file system-specific parser, and then it applies several common mutation strategies [57] (*e.g.*, bit flipping) to mutate them. If the metadata is protected by checksums, HYDRA updates them accordingly.

Syscall mutation. The goal of syscall mutation is to generate diverse, complex, and, more importantly, image-aware file operations. Similar to existing OS fuzzers [17, 23], HYDRA mutates syscall sequences in two ways: (1) altering an existing syscall in the sequence by mutating its argument(s); and (2) appending a new syscall (randomly chosen) to the end of the sequence. HYDRA also leverages the semantics of the syscall argument types to generate mostly valid syscalls, (*e.g.*, a certain range for size-like integers), to avoid being rejected early by error checking code.

Exploiting the synergy. To fuzz image and syscalls together, HYDRA schedules the two mutators in order. Specifically, given a test case, HYDRA first tries the image mutator for certain rounds. If no interesting test cases are reported, HYDRA then invokes the syscall mutator to alter arguments in existing syscalls. If still no interesting test cases are found after certain rounds, HYDRA eventually appends new syscalls.

Assisting checkers. On top of the generic strategy, HYDRA further assists bug checkers by providing checker-specific strategies. For example, when generating test cases for crash consistency testing, a valid `fsync`, or equivalent syscall is appended to generate a persistence point.

3.3 Library OS Based Executor

The traditional OS fuzzers [17, 35, 46] reuse virtualized instances for fuzzing several test cases. Because of the accumulated non-deterministic OS states, this approach, unfortunately, impedes the stable PoC generation, which developers can reproduce and debug [15, 51]. Although rejuvenating OS solves this issue, it can be as slow as a couple of seconds. Hence, HYDRA uses a library OS-based executor, which incurs negligible time (tens of milliseconds), forks a fresh instance of the executor for every test case, while consuming far fewer computing resources than VMs, enabling potentially large-scale and distributed deployment of HYDRA.

The executor serves as (1) a fuzzing target, which mounts the given image and executes the syscall trace while collecting code coverage, and (2) a bridge to checkers (§3.4, §3.5), which calls a checker, collects results, and then provides another dimension of feedbacks to the feedback engine (§3.6).

3.4 Crash Consistency Checker (SYMC3)

For various reasons (*e.g.*, performance), most file systems stage the effects of file and directory operations in memory first and flush the changes to persistent, non-volatile storage only when the time is right (*e.g.*, when system load is light). However, this optimization is not tolerable for applications that need to save critical data as quickly as possible. As a result, persistence operations, namely `sync`, `fsync`, and `fdatasync`, are used to force the in-memory states to reach the disk immediately. As a guarantee provided by file systems, any information that is flushed should be consistent even after a crash and recovery. Unfortunately, experience has revealed cases where this guarantee is violated [18, 24].

In light of this, we develop SYMC3 to vet file systems for crash consistency. Given an initial image and a syscall trace, SYMC3 emulates the syscalls to derive a symbolic representation of *all* allowed post-crash states according to the file system-specific notion of crash consistency and checks whether the recovered image falls into one of the states.

Syscall emulation. Table 2 presents a running example of how SYMC3 emulates syscalls using the bug shown in Figure 1. Mimicking the `inode` data structure in Linux file systems, SYMC3 also symbolically represents files and directories in `c3_inodes` with basic properties, *e.g.*, names, type,

#	Operation	Tree	In-memory	On-disk
0	[begin]	i0 .	i0.dents=[.]	i0.dents=[.]
1	mkdir A	i0 . i1 A	i0.dents=[., A] i1.dents=[.]	i0.dents=[.]
2	sync	i0 . i1 A	i0.dents=[., A] i1.dents=[.]	i0.dents=[., A] i1.dents=[.]
3	open A/x, O_CREAT | O_RDWR, 0666	i0 . i1 A i2 x	i0.dents=[., A] i1.dents=[., x] i2.names=[x] i2.data=[0^0] i2.size=[0]	i0.dents=[., A] i1.dents=[.]
4	pwrite A/x, "a...", 4000, 4000	i0 . i1 A i2 x	i0.dents=[., A] i1.dents=[., x] i2.names=[x] i2.data=[~~0^0,~~ $0^{4K} a^{4K}$] i2.size=[~~0,~~ 8000]	i0.dents=[., A] i1.dents=[.]
5	fdatasync A/x	i0 . i1 A i2 x	i0.dents=[., A] i1.dents=[., x] i2.names=[x] i2.data=[$0^{4K} a^{4K}$] i2.size=[8000]	i0.dents=[., A] i1.dents=[.] - i2.data=[$0^{4K} a^{4K}$] i2.size=[8000]
6	ftruncate A/x, 3000	i0 . i1 A i2 x	i0.dents=[., A] i1.dents=[., x] i2.names=[x] i2.data=[~~$0^{4K} a^{4K}$,~~ 0^{3K}] i2.size=[~~8000,~~ 3000]	i0.dents=[., A] i1.dents=[.] - i2.data=[$0^{4K} a^{4K}$] i2.size=[8000]
7	rename A/x, y	i0 . i1 A, i2 y	i0.dents=[., A, y] i1.dents=[.~~, x~~] i2.names=[~~x,~~ y] i2.data=[~~$0^{4K} a^{4K}$,~~ 0^{3K}] i2.size=[~~8000,~~ 3000]	i0.dents=[., A] i1.dents=[.] - i2.data=[$0^{4K} a^{4K}$] i2.size=[8000]
8	fsync y	i0 . i1 A, i2 y	i0.dents=[., A, y] i1.dents=[.~~, x~~] i2.names=[~~x,~~ y] i2.data=[0^{3K}] i2.size=[3000]	i0.dents=[., A] i1.dents=[.] - i2.data=[0^{3K}] i2.size=[3000]

Allowed post-crash states (POSIX):
S1) i0: ., i1: A (i2 becomes an orphan as i0 is not synced)
S2) i0: ., i1: A, i2: x, data: 0^{3K}, size: 3000
S3) i0: ., i1: A, i2: y, data: 0^{3K}, size: 3000
Allowed post-crash states (Btrfs):
S3) i0: ., i1: A, i2: y, data: 0^{3K}, size: 3000

Table 2. Symbolic representation of the c3_inode tree, and the emulated in-memory and on-disk states of c3_inodes during the execution of SymC3 on the test case in Figure 1. Boxed region represents the snapshots taken and strikethrough text maintains the history of data and metadata changes before synced.

size, link count, attributes, etc., as well as type-specific properties such as directory entries (if directory), link target (if symbolic link), and data (if regular file). However, different from the Linux inode, which only keeps the current state, the c3_inode also records the history of changes in the properties before the changes are committed to disk.

For syscalls that create an inode, such as mkdir (line 1) and open with O_CREAT (line 3), SymC3 creates a c3_inode accordingly and initializes it with proper properties, as in the case of i1 and i2 in the example. SymC3 also creates a snapshot of the c3_inode tree, indicating that ./A and ./A/x might exist on disk if the crash occurs after the execution. Similarly, for syscalls that manipulate the tree structure, such as rename (line 7), a snapshot is created to reflect the fact that the ./y might exist on disk if the effect of rename has reached the disk. However, no snapshot allows the existence of both ./A/x and ./y. Furthermore, regardless of whether x or y is persisted, it should map to i2.

According to the POSIX standard, among syscalls that persist inode, sync and fsync commit the entire c3_inode, *i.e.*, both data and metadata, to disk (line 2 and 8, respectively) while fdatasync only commits data and those metadata that are related to data (*e.g.*, size, checksum) to disk (line 5). Other syscalls (line 4, 6) modify either data or metadata of the c3_inode, and the changes are piled and versioned in memory until reaching a persistence point. For example, after fdatasync in line 5, changes in the i2 data and size are committed to disk and their prior versions (*i.e.*, empty file with zero size) can be safely discarded, as from now on the disk is no longer allowed to recover to the previous state. In other words, SymC3 keeps track of the change history for each c3_inode property until they are persisted.

Enumerating crash states. At any stage, SymC3 is capable of generating the set of allowed post-crash states by enumerating the snapshots along with on-disk and in-memory c3_inodes. For each c3_inode tree snapshot, SymC3 checks whether it meets the requirement that if a c3_inode is known to be persisted, (*i.e.*, directory entry exists on disk), the snapshot must contain it. With this constraint, SymC3 rules out invalid snapshots. In Table 2, among the four snapshots, the first tree, which has i0 only, is dropped because of the constraint: i0 and i1 are known to be persisted, but i1 is not in the snapshot. The other three do not violate any constraint.

With the valid sets of snapshots, SymC3 further multiplexes them with the allowed states per each c3_inode property to generate *all* crash-safe states. In the running example, all c3_inodes in each valid snapshot are persisted, leading to one possible state per snapshot and making the total allowed states to be three. If SymC3 finds that the recovered image does not fall into any of the allowed states, a bug is reported. SymC3 reported the running case because after the crash, Btrfs recovered the image to the state where the size of file y was 8000, which is not one of the allowed states.

This test case is particularly interesting in that when running the case on the patched kernel, Btrfs and F2FS yield different crash states. Btrfs recovers to **S3**, while F2FS recovers to **S2**. Since SymC3 considers both as legal states, neither is considered a bug. This is in contrast to the design choice made by B3 where it only considers the final snapshot before the crash, namely **S3**, as a correct oracle state, which is in fact only a subset of all possible states. As further supported by our experiment in §5.6, this could be the main reason B3 incurs a high false positive rate.

Extending consistency semantics. POSIX specification itself is "loose," leaving much room for implementation-specific behaviors in handling crash consistency. As a remedy, Hydra implements various file system-specific consistency semantics to conservatively handle stronger crash guarantees. For example, although not mandated by POSIX,

Btrfs persists the *directory entry* as well as metadata when an inode is fsynced. This is why the renamed file y is persisted without explicitly fsyncing its parent directory in the example case. In addition, ext4, Btrfs, and F2FS resolve the symbolic path to the original file/directory if it is provided as an argument, although the POSIX entry for link states that the behavior is implementation-defined.

Apart from the extensions, another aspect to consider is that not all file systems implement POSIX fully. For example, the FSCQ's specification regarding unlink deviates from the POSIX standard because FSCQ relies on Filesystem in Userspace (FUSE) driver; it does not allow unlink to be conducted on an open file, even though POSIX states that "*if one or more processes have the file open when the last link is removed, the link shall be removed before unlink() returns.*" Handling such deviation in SymC3 is not difficult, because we only need to enforce these additional rules on top of the generic file system layer that is emulated. These extensions require only a few lines of changes (Table 3).

3.5 Other Checker Plugins

Besides the home-grown consistency checker SymC3, we further show that three independently developed checkers – each of which targets a different type of file system bug and is *not* originally designed for fuzzing – can be seamlessly plug-and-played in Hydra with little or no engineering effort.

POSIX conformance. We integrate SibylFS [43] in Hydra to find POSIX violations in file systems. SibylFS formalizes the range of POSIX-allowed behaviors of a file system for any sequence of syscalls within its scope. Based on these formalizations, SibylFS serves as an oracle to decide whether an observed trace is allowed by the POSIX specification given the initial image and the sequence of syscalls. To bridge Hydra with SibylFS, we run the SibylFS oracle in a standalone process and connect it with the bug-checking dispatcher via a dedicated channel. Whenever the executor receives a test case, the dispatcher forwards the test case to the SibylFS process, which handles test case unpacking and oracle checking and eventually replies with a signal on whether it detects any violations of the POSIX standard.

Logic checking. As noted in §2.1, most logic bugs cause silent failures and finding such bugs often requires hooking the file system at runtime with domain-specific invariant checks. Although file system developers are often aware of this issue and have placed precautionary checks in the codebase, the checks are rarely enabled for performance reasons. This makes them a perfect match for fuzzing, as Hydra aims to find ways to trigger these assertions that are otherwise missed in normal workloads. Most developer-annotated checks can be conveniently integrated with Hydra by specifying the corresponding CONFIG_* options when compiling the target file system, *e.g.*, CONFIG_BTRFS_FS_REF_VERIFY for the Btrfs extent tree reference verifier [1]. The dispatcher monitors any warnings or errors raised from these checks and accordingly marks the test case as interesting.

Memory safety. Although not the focus of Hydra, to be complete, Hydra also looks for memory errors leveraging the Kernel Address Sanitizer (KASan) [13], especially out-of-bound and use-after-free bugs in file system implementations. Whenever KASan reports an error at runtime, the bug-checking dispatcher of Hydra crashes the kernel execution and marks the input test case as interesting.

3.6 Feedback Engine

In Hydra, a test case execution is summarized in the feedback report, which essentially measures the "novelty" of a test case and decides whether it deserves further mutation. Hydra considers two types of feedback:

Branch coverage. Like traditional fuzzers, for Hydra, a file system is represented as a control-flow graph where vertices are basic blocks and edges are branches from one basic block to another. While executing a test case, Hydra keeps track of the set of edges visited, and the novelty of the test case is measured by the number of new branches and unique combination of branches triggered. By default, branch coverage forms the primary coverage metric, which Hydra leverages to find bugs of any type.

Checker-defined signal. As a generic fuzzing framework, Hydra's feedback engine additionally allows each checker to register its own feedback formats. In its simplest form, as used by all checkers in Hydra, the feedback is just a boolean variable indicating whether or not this test case triggers a buggy condition (*i.e.*, 1 for bug, 0 otherwise). For a more sophisticated example, a checker that tries to uncover specification violations may provide a feedback format that tracks the number of rules that have already been asserted by prior runs in the given specification. This will penalize input mutators for generating test cases that cover the asserted parts and will eventually drive Hydra towards the yet-to-be-tested part of the specification.

3.7 Post Bug Processing

Most fuzzers stop at finding an erroneous state without worrying about reproducibility and the size of the test case. The assumption is that debug information (e.g., KASan reports or stack traces) can help locate the bug. This assumption might be valid for small applications. However, for file systems, in most cases, the debug information reveals only the direct symptom instead of the root cause, and even the file system maintainers need to spend a day or two navigating through hundreds of syscalls to pinpoint the root cause, as shown in the piled-up bugs in syzbot [15]. In light of this, Hydra takes the extra steps to reproduce each found bug in a VM with realistic kernel and file system settings and minimize the corresponding test case into a suitable-sized PoC.

VM-based replay. A library OS running in user space typically has its own implementation of scheduling, memory management, and interrupt handling, which are largely different from those of the original Linux kernel. For example, some library OSes (e.g., Linux Kernel Library [40]) only support single threading, and the execution order of the kernel code in such a library OS differs from that of a machine that supports multi-threading. Therefore, Hydra relies on real VM instances to verify that every found bug does indeed affect the end users of the tested file system. When a bug is found, Hydra replays the corresponding test case on a fresh VM instance, which is installed with the same kernel and file system used by Hydra's libOS-based executor. The bug is confirmed when the replays result in the same runtime violations captured by Hydra during the fuzzing process.

Test case minimizer. For file system developers, an ideal PoC should be minimal, *i.e.*, the syscall sequence should only include the necessary operations that trigger the bug. Unfortunately, the raw bug-triggering test cases generated by Hydra are far from minimal, with excessive mutations on the syscall sequence that have no effect in bug manifestation. To reduce a raw PoC into a minimal PoC, Hydra uses the Delta Debugging technique [58]. To be specific, with the given syscall sequence, Hydra tries to remove one syscall and retest until a minimal syscall trace is reached, *i.e.*, removing any one in the trace voids the bug. Although this approach is still sub-optimal compared with synthesizing test cases from scratch, it is highly effective in practice and can be further improved by advanced syscall distillation techniques [20, 36].

4 Implementation

Hydra adopts the basic fuzzing infrastructure from AFL 2.52b [51, 57], including the fork server, code coverage bitmap, and test case scheduling, but replaces a few key components, including an input explorer that mutates both file system images and syscalls. We leverage the file system-specific utilities (*e.g.*, `mkfs` and `fsck`) available in development packages (*e.g.*, `e2fsprogs`) to identify the metadata chunks of each file system. We also use these utilities to inspect the image after mutation: *i.e.*, to iterate files and directories on the image and feed information for the syscall mutator to generate image-aware syscall sequences (see, Table 3).

We choose the Linux Kernel Library (LKL) [40] as the library OS for the executor. The official LKL is based on Linux kernel v4.16 and we ported it to v5.0. When compiling LKL, we restrict instrumentation (*e.g.*, code coverage tracking for AFL) to the tested file system only; therefore, we can focus only on the exploration of the file system instead of on the whole kernel. LKL is statically linked into the executor, which takes the input from the mutator, mounts the image, and runs the syscalls by calling LKL functions.

Component	LoC	Language
Hydra Framework		
Input mutators	8,507	C++ / Python
LKL-based executor	2,190	C++
Glues to checkers	233	C / C++ / Python
Feedback engine (as AFL changes)	497	C
Bug post-processing	274	Python / Bash
Crash Consistency Checker (SymC3)		
Syscall emulator & violation checker	3,349	Python
`ext4` extension	3	Python
`F2FS` extension	3	Python
`Btrfs` extension	35	Python
FSCQ extension	4	Python

Table 3. Implementation complexity of Hydra and SymC3.

Despite the large shared codebase on syscall emulation and violation checker, SymC3 is extremely flexible in incorporating customized notions of crash consistency or non-POSIX compliant operations in various file systems. For example, the crash consistency property in `Btrfs` requires it to persist the directory entry as well as metadata when an inode is `fsync`-ed. This deviation from the standard behavior can be modeled in as small as 35 LoC (see, Table 3).

5 Evaluation

We evaluate Hydra by fuzzing popular, heavily tested (even formally verified) Linux file systems. In particular, we show the effectiveness of Hydra with the number of new bugs discovered by various checkers in Hydra (§5.1). The effectiveness can be explained from three aspects: (1) a high fuzzing speed allows Hydra to explore file system states quickly (§5.2); (2) the dual-aspect input mutation allows Hydra to explore more execution paths than both existing OS fuzzers and specialized bug checkers (§5.3); and (3) the addition of checker feedbacks allows Hydra to further lean its fuzzing effort towards checker-defined states besides greedy path exploration (§5.4). Beyond hitting more bugs, we further evaluate Hydra's performance in bug reproducibility and test case minimization (§5.5). Additional evaluations on the in-house developed crash consistency checker are performed to support how it fares against prior works (§5.6).

Experimental setup. We run Hydra on a 2-socket, 24-core machine running Ubuntu 16.04 with Intel Xeon E5-2670 and 256GB RAM. We tested `ext4`, `Btrfs`, and `F2FS` in Linux kernel v5.0, and also FSCQ (`sosp17` branch). We also tested `XFS`, `GFS2`, `HFS+`, `ReiserFS`, and `VFAT`, but found only memory-safety bugs. Unless otherwise stated, we use a minimal seed disk image that contains seven different types of files and directories (e.g., hard/soft links, `FIFO`, `xattr`, etc.) in all fuzzing runs. We compare Hydra with the latest version of two state-of-the-art OS fuzzers, Syzkaller and kAFL. Syzkaller runs with the KVM instances, each of which has two cores and 2GB RAM.

5.1 Bug Hunting in Popular File Systems

Across intermittent runs during a 10-month period of development, Hydra discovered 91 new bugs in total, of which 60

FS	Crash Inconsistency		Logic Bug		Spec. Violation		Memory Error	
	#A/#R/#F	T	#A/#R/#F	T	#R	T	#A/#R/#F	T
ext4	1 / 1 / 0	4w	0 / 0 / 0	1w	1	1w	3 / 4 / 3	1w
Btrfs	1 / 4 / 1	8w	7 / 7 / 3	1w	2	1w	21 / 21 / 12	1w
F2FS	2 / 3* / 2*	4w	16 / 16 / 16	1w	1	1w	8 / 8 / 8	1w
FSCQ	1 / 1 / 1	1w	- / - / -	-	-	-	- / - / -	-
Total	5 / 9* / 4*	17w	23 / 23 / 19	3w	4	3w	32 / 33 / 23	3w
VM-replay	9/9 (100%)		22/23 (96%)		4/4 (100%)		30/33 (91%)	

A: **A**cknowledged, **R**: **R**eported, **F**: **F**ixed, **T**: **T**ested time (in weeks)

Table 4. Hydra found 91 new bugs (69 in the four file systems listed above and 26 in others) along with four POSIX violations. Hydra successfully reproduced all cases except four on VM (see, §5.5). As the bugs in other file systems are memory errors while the focus of Hydra is semantic bugs, we omit them in this table. We acknowledge that the found POSIX violations are known to the Linux communities, so they remain unfixed. *One of the crash consistency bugs we found in F2FS was fixed before we reported.

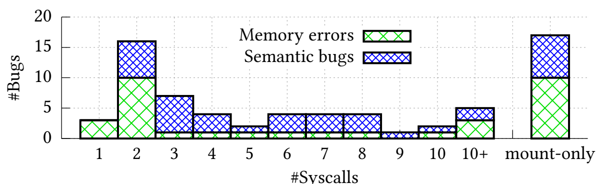

Figure 7. The number of syscalls required to trigger each file system bug found by Hydra after minimization.

have been confirmed and 46 bugs fixed (see, Table 4). Note that we also found four POSIX violations. The results show that as a fuzzer tailored to file systems, Hydra, together with the checker plugins, helps us discover and patch a diverse set of bugs in various file systems. More importantly, among all bugs found, 32 are semantic bugs that do not cause kernel panics when triggered, and hence, cannot be found by prior OS fuzzers. This sheds light on how to stress not just memory errors but also the semantic parts in file systems by integrating specialized checkers into the Hydra fuzzing framework. Note that even though the memory bugs found seem to outnumber the semantic bugs, it does not impair our claim that Hydra is a generic and effective framework applicable to many bug types, including hard-too-detect semantic bugs (*e.g.*, the crash inconsistency found in verifed FSCQ). Besides, KASan is another checker plugged into Hydra, supporting the manifestation of memory bugs.

To see how deeply Hydra explores the input spaces of a file system, we further measure the minimal trace of syscalls (obtained from the test case minimizer) required to trigger these bugs. The result is shown in Figure 7, and we make several observations:

Dual-aspect input mutation. Seventy-five percent of found bugs require both mounting a crafted file system image and subsequently executing a dedicated sequence of syscalls to trigger. This shows the importance of exploiting the synergy of two types of mutations. The rest of the bugs (7 logic bugs and 10 memory errors) can be triggered by merely mounting

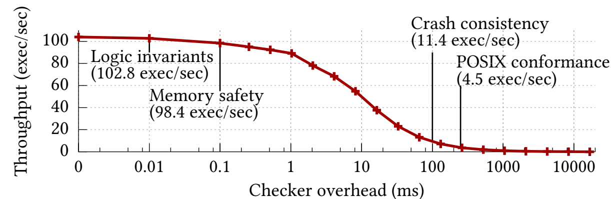

Figure 8. Performance of Hydra's state exploration with checkers exhibiting different overheads, indicating the expected throughput when a developer integrates a custom checker with Hydra.

a corrupted image, which shows the importance of image mutations.

Bug complexity. A file system bug can be triggered by as little as one syscall[2] or as many as 32 syscalls after minimization. Although there is no rule-of-thumb for how many syscalls are enough to reach a file system bug of a specific type, we do observe that semantic bugs tend to require a longer sequence of syscalls to be triggered compared with memory errors. In fact, among all the bugs that require file operations to manifest, 57% of memory errors can be triggered with just one or two syscalls, while the ratio is only 21% for semantic bugs. In contrast, nearly 50% of semantic bugs require at least five syscalls to reach. Therefore, it might not always be valid to assume that small workloads are sufficient to reveal file system bugs of all types. Testing a file system with a fixed length of random syscalls (*e.g.*, B3 seq-3) might miss many bugs to be found.

5.2 Fuzzing Speed

The overall throughput of Hydra highly depends on the speed of a companion checker; intuitively, a more expensive checker implies lower throughput (Figure 8). For example, checking POSIX violation is the most expensive, as each test case needs to be piped to a separate process that hosts an OCaml runtime and re-emulates the syscalls with extensive specification checking. On the other hand, the logic checker places accounting and assertion hooks inlined with the code, allowing Hydra to explore the input space at full speed.

It is worth noting that the performance of a checker is not a limitation, as Hydra is not responsible for reducing the checking overhead. Rather, we consider that the amount of analysis (and hence the overhead) is the price we have to pay to find bugs of a particular type. The theoretical maximum is 122 exec/sec, measured by merely starting and stopping the LKL instance. The difference between this and the Hydra baseline (104 exec/sec) reflects the overhead of the generic fuzzing infrastructure, including input mutations and bookkeeping (*e.g.*, updating the AFL coverage bitmaps).

For comparison, a VM-based approach takes at least 1.4 sec (0.7 exec/sec) to achieve the same effect, *i.e.*, a clean-slate kernel and file system for every test case, which is 100× slower than our libOS-based executor. In addition, although the

[2] A bug in `ext4` can be triggered with only one syscall: `removexattr("A/B/acl", "system.posix_acl_access")`.

throughput might be low for expensive checkers, this may be compensated by paralleling Hydra, similar to distributed fuzzers like syzbot [15].

5.3 Code Coverage

Besides throughput, code coverage, especially the coverage when fuzzing is toward saturation, is another important factor that decides the effectiveness of fuzzing. Intuitively, the more execution paths covered, the more thoroughly a file system is tested. Hydra achieves higher code coverage than not only existing OS fuzzers but also bug checkers that synthesize test suites with their own algorithms.

Comparison with existing OS fuzzers. We perform controlled experiments on Hydra and state-of-the-art OS fuzzers Syzkaller and kAFL by 1) mutating file system images only and fuzzing with the same sequence of syscalls, 2) mutating syscall sequences only and fuzzing with the same seed image, and 3) mutating both. The results are shown in Figure 9. At the end of the 12-hour period, Hydra outperforms Syzkaller by 1.55x, 1.52x, and 1.45x for `ext4`, `Btrfs`, and `F2FS`, respectively, and outperforms kAFL by 8.74x, 6.31x, and 6.47x.

By mutating image metadata only, upon saturation, Hydra_i explores at least 3.36x code paths in all tested file systems compared with both Syzkaller_i and kAFL. This is because Hydra identifies metadata chunks in an image with file system-specific parsers, while Syzkaller_i mutates the non-zero parts only and kAFL mutates the first 2K bytes only. Both Syzkaller_i and kAFL may miss important metadata chunks as well as include non-essential user data. For syscall mutation, Hydra_s is still slightly better than Syzkaller_s (at most 1.12x). The improvements mainly come from generating image-aware workloads.

On top of that, Hydra achieves higher code coverage than both Hydra_s and Hydra_i, which proves the importance of dual-aspect input mutation in file system fuzzing. Moreover, Hydra also outperforms Syzkaller on all tested file systems. Having more effective mutators is one reason. More importantly, Hydra wisely schedules two mutators (see §3.2) while Syzkaller does not prioritize either of them, leading to even worse performance than Syzkaller_s in all cases.

Comparison with synthesized test suites. To support the claim that bug checkers can offload the path exploration component to Hydra, Hydra should be additionally compared with test synthesizers in these checkers, as these synthesizers share the same goal with fuzzing: exploring as many program states as possible. To this end, we check whether the test cases generated by Hydra yield more code coverage than those synthesized by individual bug checkers recently proposed; the results are shown in Figure 10.

B3 seq-2 test suite. Even though we restrict the syscalls to be the same set as supported by B3, Hydra yields more coverage than B3's seq-2 test cases. This is because B3 selects arguments from a pre-defined small set when generating test cases. For example, `truncate`'s length is either 0 or 2500, and only two directories and two files are used as path arguments. On the other hand, Hydra mutates these arguments, discovering more code paths.

FS	Crash Inconsistency		Logic Bug		Spec. Violation	
	W/ Sig	No Sig	W/ Sig	No Sig	W/ Sig	No Sig
`ext4`	0	0	0	0	1	1
`Btrfs`	4	2	5	1	2	2
`F2FS`	2	1	8	2	1	1
Total	6	3	13	3	4	4

W/ Sig: With Signal, **No Sig**: No signal

Table 5. Effectiveness of checker feedbacks. The table shows how many unique bugs are found with checker feedback and by replaying the test cases generated without checker feedback.

SibylFS test suite. Similarly, Hydra yields more coverage than the test suite synthetized by SibylFS even when fuzzing is restricted to the same set of syscalls. One reason is that SibylFS heavily relies on the manual enumeration of equivalence classes, which can be incomplete. For example, SibylFS seems to treat `read(fd, buf, 100)` and `read(fd, buf, 10000)` as equivalent, because both reads from the same file descriptor, while they may trigger different code paths as the length crosses the page and block size. Another reason is that SibylFS focuses more on enumerating combinations of arguments for a single syscall, generating less diverse syscall sequences.

5.4 Checker Feedback

For semantic bug checkers, besides greedy exploration of code paths, what is equally important if not more valued is the triggering of their "favored" states, *i.e.*, the states in which the checking actually occurs and errors are likely to be reported. For example, for checkers that enforce developer-annotated invariants, what developers hope for is exploring not only more paths in the file system, but also more paths that go through the annotated checks. Hydra supports this by adding feedback from the checker to the input mutator. The feedback can be as simple as a boolean variable indicating whether or not the checker favors this test case, which is the case in SymC3, which sends 1 when it finds a crash inconsistency. Intuitively, by sending positive feedback, the checker expresses its intention to see more inputs like this.

We show this with a controlled experiment. We first run Hydra with checker feedback for 12 hours and collect the number of test cases flagged as buggy by the checker. We then run Hydra without checker feedback for 12 hours[3] and collect all the test cases AFL saved in the seed pool. With only branch coverage feedback, the seed cases represent the situation where the fuzzing effort is not directed toward any particular states. The last step is to re-run these seed cases with checker enabled again and see how many unique bugs are found. The results are shown in Table 5.

[3] In this run, Hydra still invokes the checker, but regardless of whether a bug is reported, the feedback is ignored. This allows the execution to proceed given semantic bugs are unlikely to cause visible impact.

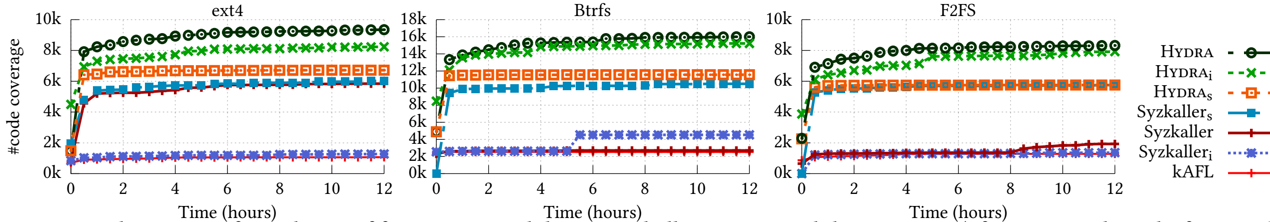

Figure 9. Code coverage for 12 hours of fuzz testing with kAFL, Syzkaller, HYDRA and their variants (i for image-only and s for syscall-only mutation). It indicates that HYDRA aggressively explores the input space that both kAFL and Syzkaller could not reach, and combining both image and input mutators exhibit a synergistic effect in exploring different parts of the input space.

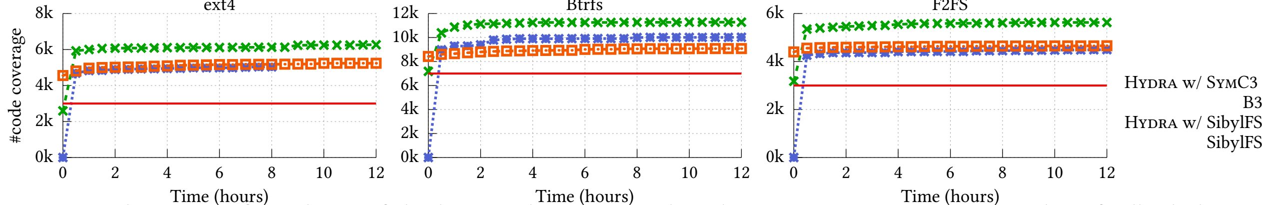

Figure 10. Code coverage for 12 hours of checking crash consistency bugs by using a static mutator, B3, and our feedback-driven fuzzer, HYDRA with SYMC3, as well as checking specfication violation using SibylFS's test suite versus HYDRA's dynamically generated test cases. In finding both bug types, not only does HYDRA visit more code paths but also keeps discovering new paths throughput execution—as the exploration quickly saturates, finding just one new path for a few hours of executions is a significant indicator. Note that B3's seq-2 test cases for `ext4` complete in 8 hours (the leftmost graph), and SibylFS's test suite completes within an hour (the straight red lines).

On average, disabling the feedback means that we will miss 57% of the semantic bugs that could be caught with the checker feedback. The explanation lies in the seed-selection policy. In particular, seeds receiving positive feedback from the checker are prioritized for more mutations. The rationale is that a test case favored by the checker will likely have some erroneous states accumulated. Therefore, by exploring more along that direction, HYDRA has a higher chance to trigger more bugs. However, this effect does not show up for POSIX violations, as these bugs are too shallow to be missed.

5.5 HYDRA Framework Services

HYDRA provides bug processing as a framework service to all bug checkers in an attempt to address the challenge in traditional OS fuzzing: irreproducible bugs due to the accumulated effects of thousands of syscalls. We measure how successfully HYDRA achieves its goal from two aspects:

VM-replay for bugs found. Whenever the bug checker flags a test case, HYDRA replays the test case on a fresh VM instance running with the same kernel and file system HYDRA uses for fuzzing. We manually check whether the VM replays the same behavior as shown in the LKL executor, *i.e.*, (1) being in an inconsistent state (for crash inconsistency), (2) deviating from standards (for POSIX violations), (3) failing at assertions (for logic bugs), or (4) panicking with the same KASan or `BUG()` location (for memory errors). The results are summarized in the last row of Table 4.

Only four bugs are not always replayable (although they are acknowledged by the developers), and the reason is that LKL and the kernel in the VM use different schedulers. As a result, these timing-critical bugs do not always manifest in the VM. For instance, an `ext4` memory error is caused by the `JBD2` thread running in the background. Since LKL is a uniprocessing and non-preemptive kernel, we can trigger the bug, as syscalls and the `JBD2` thread are serialized but not on the VM where we have no control of the scheduling.

#	FS	#Syscall (min.)	B3 sequence
1	`ext4`	36⟶ 3 (91.7%)	seq-1★ (requires support for chmod)
2	`Btrfs`	164⟶12 (92.7%)	seq-5
3	`Btrfs`	151⟶ 6 (96.0%)	seq-1★ (opens and persists additional file)
4	`Btrfs`	44⟶ 6 (86.4%)	seq-3★ (mix of data and metadata ops)
5	`Btrfs`	40⟶ 8 (80.0%)	seq-4
6	`F2FS`	233⟶ 6 (97.4%)	seq-2★ (writes on overlapping regions)
7	`F2FS`	20⟶ 3 (85.0%)	seq-1★ (requires support for chmod)
8	`F2FS`	29⟶ 8 (72.4%)	seq-3★ (opens and persists additional file)
9	FSCQ	36⟶ 7 (80.6%)	seq-2

Table 6. The number of system calls used to trigger new bugs, before and after minimization. Even after minimization, all test cases, except FSCQ bug, cannot be reached by B3's input generator. ★ indicates that some relaxation of boundaries is required: *e.g.*, #bug3 requires one core operation (*i.e.*, `chmod`) but needs to open and persist another file, which is not reachable by B3's seq-1 workloads.

Test case minimization. We also evaluate whether HYDRA is capable of eliminating syscalls that do not contribute to the manifestation of the bug. We run the minimizer on all bugs HYDRA found, and on average, the minimizer reduces the number of syscalls in the PoC from 47.5 to 5.6, yielding an 88% reduction. As a snippet of the effectiveness of the minimizer, Table 6 shows how it reduces the syscalls in the PoC for the nine crash consistency bugs HYDRA found.

5.6 Crash Consistency Checker (SYMC3)

We compare the crash consistency checker of HYDRA with the state-of-the-art prior work, B3, in detail, focusing on false positives and performance.

Correctness. We tested HYDRA with 26 previous crash consistency bugs that B3 collected as well as 10 new bugs that B3 found. SYMC3 detected 24 out of 26 previous bugs. One bug

	B3					Hydra
	Type	#Tests	Size	T-Gen	T-Comp	(w/ SymC3)
Prep.	seq-1	0.3k	2.8 MB	<1m	<1m	None
	seq-2	240k	2.7 GB	3h 28m	6h 31m	
	seq-3	8,241k	94.8 GB	5d 1h	-	
Exec	0.2 exec/s					11.4 exec/s
FP+	31k (100%)					0 (0%)

T-Gen: Time to generate, **T-Comp**: Time to compile

Table 7. Comparing Hydra with B3 in terms of execution time and precision in testing Btrfs of Linux v4.16. Hydra generates test cases on the fly, while B3 requires extensive resources for preprocessing When testing seq-1/2 test cases, Hydra incurs no false positives, while B3 reports 31k incorrect consistency errors.

missed by SymC3 requires a special command, `dropcaches`, and B3 also missed the same bug. Another bug requires `msync`, which SymC3 currently does not support.

SymC3 successfully detected all 10 bugs newly discovered by B3. However, B3 missed *all* the bugs Hydra newly discovered, for the following four reasons: (1) requiring more than three core operations, (2) requiring the combining of both metadata and data operations[4], (3) not supporting crucial file system operations, *e.g.*, `chmod`, or (4) not supporting different file types *e.g.*, `FIFO` file.

False positives (incorrect reports). B3 runs the same test case on two empty file system images while keeping one image as a reference oracle and crashing the other. Then, it tests crash consistency by comparing the files and directories in the oracle with those in the recovered image. As noted in §3.4, the oracle is *one* of the possible post-crash states, and B3 ends up flagging legitimate cases as bugs. Because of this systematic limitation, B3 reported 31k incorrect consistency errors from the `Btrfs` file system in the v4.16 kernel, while Hydra raised no false positives. The only possibility of SymC3 having false positives is through bugs in the checker. However, we fuzz-tested SymC3 for a month by running SymC3 without injecting a crash condition in the LKL executor until there were no more bugs to fix.

Performance. B3's strategy of enumerating all input space requires a considerable amount of space and preprocessing time for generating and compiling the test cases. As shown in Table 7, it required more than five days for B3 to generate 8M seq-3 test cases. To make matters worse, much bigger input space, *i.e.*, up to seq-5, has to be enumerated to generate the test cases that cover the new bugs found by Hydra, which makes B3 infeasible. On the other hand, because of fuzzing, the input generation in Hydra is dynamic and requires no preprocessing. In addition, the speed of execution of Hydra is orders of magnitude faster than that of B3, further showing Hydra's usability as a large-scale framework.

[4]B3 only generated three non-exhaustive sets of seq-3 test cases: *seq-3-data*, using write operations only, *seq-3-metadata*, using metadata operations only, and *seq-3-nested*, using link and rename on nested directories and files.

```
1 fd_root = open(".", O_DIRECTORY, 0);
2 fd_foo = open("./foo", O_CREAT | O_RDWR, 0777);
3 fsync(fd_foo);
4 mkdir("./A", 0777);
5 ftruncate(fd_foo, 5595);
6 pwrite64(fd_foo, buf, 4000, 1303);
7 fsync(fd_root);  // should persist both A and foo
```

Figure 11. FSCQ- `fsync` fails to persist directory `A` after `fsync` on root directory. After a crash and recovery, only `foo` is in the file system.

```
1 fd_root = open(".", O_DIRECTORY, 0);
2 fd_foo = open("./foo", O_CREAT | O_RDWR, 0777);
3 fsync(fd_root);
4 write(fd_foo, buf, 4000);
5 fdatasync(fd_foo);  // foo should be size 4000
```

Figure 12. FSCQ- `fdatasync` fails to persist data in file `foo`. After a crash and recovery, `foo` is empty.

6 Case Study: Bugs in FSCQ

FSCQ [7] is a formally verified file system with proven specifications regarding correct behaviors of a crash-safe file system, especially the precise definition of `fsync` and `fdatasync` and how they react to logged writes and log-bypassing writes. However, to our surprise, Hydra is still able to find two bugs that violate the crash consistency property, and both have been acknowledged by FSCQ developers.

Figure 11 shows the PoC when `fsync` fails to persist a directory entry. This bug triggers an uncovered part in their proof, as noted by the developer,

> *the design of DFSCQ should permit them to be crash safe, but the proofs don't cover mixing direct and logged writes, and logged writes currently are not synced correctly.*

The `pwrite64` syscall (line 6) accidentally triggers the mixing of direct and logged writes. As a ballpark fix (commit `97b50e`), all the logged writes in the `fscqWrite` procedure are disabled. This case proves the effectiveness of Hydra in generating test cases that are hard for developers to contemplate even with the help of machine-checked proofs.

Figure 12 shows the PoC when `fdatasync` is not persisting data written to files. According to FSCQ developers, this is due to different interpretations of the `fdatasync` on the Linux man page [27], especially the part on "*`fdatasync`... to allow a subsequent data retrieval to be correctly handled.*" If one interprets "correctly handled" as all previously written data to the file should be readable, this is a bug. However, this is not the specification FSCQ formalizes. In FSCQ, `fdatasync` forms a weaker guarantee: either empty content or the previously written data in its integrity is allowed, but nothing in between. In this case, either 0 or 4000 as the size of `foo` is allowed. In FSCQ, to force data to touch disk, `fsync` is required. This bug is found and reported by B3 as well; therefore, we do not count it in the new bugs found. By updating SymC3 to adopt the notion taken by FSCQ developers, SymC3 can tolerate this relaxed interpretation of `fdatasync`.

7 Discussion

Novelty over Janus. Although Hydra has a mutation strategy similar to Janus, there is a prominent difference between these works. Unlike Janus which *focuses only on memory errors*, Hydra allows developers to find deeper semantic bugs that have more devastating effects, such as data loss (crash inconsistency) and programmable errors (inconsistent states and specification violations). At a high level, Janus is only a special instance of Hydra and Hydra generalizes Janus's approach, *i.e.*, multi-dimensional exploration, and broadens the scope of file system fuzzing by incorporating an additional dimension: bug checkers.

Value as a Framework. As a framework, Hydra provides an automated process of revealing in theory any *developer-defined buggy situations*: on behalf of developers, Hydra takes charge of automated input exploration, checker incorporation, and validation of found bugs while the bug checker is only responsible for accurate description of the bug. Such separation of concerns drastically improves the quality of bug finding: higher efficiency and lower false positives. As shown in §5.6, by letting developers solely focus on writing a precise checker for crash inconsistency, Hydra could find bugs with higher accuracy and in less time than B3.

8 Related Work

Hydra is the first generic fuzzing framework that is capable of testing and finding various types of bugs from existing file systems. This section introduces and compares the most relevant prior work with Hydra.

Finding bugs in file systems. There are two broad categories of approaches to find or eliminate bugs in file systems, namely, model checking and formal verification.

1) Model checking can be done in a static or dynamic fashion. Static approaches, such as JUXTA [31], and FERRITE [5], attempt to compare either well-specified [5] or inferred [31] models with the design or implementation of file systems. Since static checkers lack concrete execution states, they fundamentally suffer from high false positives, *e.g.*, having stochastic errors in inferred models [31].

Dynamic approaches, such as FiSC [54] and eXplode [52] for general storage system bugs, B3 [33] for crash consistency, and SibylFS [43] for specification violation, concretely run and validate test cases, thereby rendering high true positives (*i.e.*, most reported bugs are real). However, unlike Hydra, which directs the input exploration toward a targeted bug type while testing, existing methods aim to check the model against a fixed but too-humongous-to-explore number of test cases, exhaustively testing all the possible non-deterministic executions.

2) Formal verification [7, 8, 28] is a promising new direction that can, in theory, eliminate all the semantic bugs of file systems. However, as demonstrated by Hydra, formal verification in practice suffers from incorrect assumptions in the proofs or often must embed unverified code such as interface or driver. Hydra can complement its effort by checking bugs on the whole, end-to-end system as it is after the verification.

Fuzzing beyond memory safety. Besides the memory safety bugs, there have been works that applied fuzzing techniques to find other types of bugs, such as race condition [22], use-before-initialization vulnerabilities [29], or even bugs in deep learning systems [37]. As described in §3.5, Hydra can readily be extended to find these bug types by providing corresponding checkers for existing file systems.

Fuzzing kernels. Syzkaller [17], kAFL [46], TriforceAFL [35] are the state-of-the-art fuzzing frameworks for kernels. Unfortunately, their focus is to find non-semantic bugs such as memory errors that have a clear signal, and thus fail to trigger deep semantic bugs in file systems. In addition, by specializing our focus to file systems, Hydra can shorten the execution time of a single test case, as well as increase the reproducibility of found bugs by running the file system code in user space with libOS.

9 Conclusion

This paper presents Hydra, an extensible fuzzing framework to find in theory any types of bugs in file systems. Hydra cleanly separates the process of exploring the input space from validating the existence of bugs of interest. Thus, with Hydra, developers may now focus on the core logic for hunting bugs of their own interest, while Hydra takes care of file system state exploration and test minimization. In our prototype, we equip Hydra with both home-grown and external bug checkers and discovered nine crash inconsistencies, four POSIX violations, 23 logic bugs, and 59 memory errors across various Linux file systems, including the verified FSCQ. In particular, our crash consistency checker, SymC3, outperforms state-of-the-art checkers in both accuracy and performance. With existing and future file system checkers unified under one umbrella, Hydra can be the go-to solution for one-stop testing on multiple aspects of file systems to improve their quality. Looking forward, besides integrating more checkers for local file systems, Hydra may be further extended for fuzzing networked and distributed file systems.

10 Acknowledgment

We thank Tej Chajed, Theodore Ts'o, the anonymous reviewers, and our shepherd, Junfeng Yang, whose comments helped improve the paper. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851, and CNS-1749711; ONR under grant N00014-18-1-2662, N00014-15-1-2162, and N00014-17-1-2895; DARPA TC (No. DARPA FA8650-15-C-7556); ETRI IITP/KEIT[B0101-17-0644]; and gifts from Facebook, Mozilla, Intel, VMware, and Google.

References

[1] Josef Bacik. 2017. Btrfs: add a extent ref verify tool. https://patchwork.kernel.org/patch/9978579/. (2017).

[2] Wendy Bartlett and Lisa Spainhower. 2004. Commercial Fault Tolerance: A Tale of Two Systems. (2004).

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[5] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, 83–98.

[6] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. 2007. Ext4: The Next Generation of Ext2/3 Filesystem.. In *USENIX Linux Storage and Filesystem Workshop*.

[7] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-performance Crash-safe File System Using a Tree Specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[9] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. 2004. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*. San Francisco, CA.

[10] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.

[11] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, California, USA.

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[13] Google. 2016. KernelAddressSanitizer, a fast memory error detector for the Linux kernel. https://github.com/google/kasan. (2016).

[14] Google. 2018. KernelMemorySanitizer, a detector of uses of uninitialized memory in the Linux kernel. https://github.com/google/kmsan. (2018).

[15] Google. 2018. syzbot. https://syzkaller.appspot.com. (2018).

[16] Google Inc. 2015. KernelThreadSanitizer, a fast data race detector for the Linux kernel. https://github.com/google/ktsan. (2015).

[17] Google Inc. 2019. Syzkaller is an Unsupervised, Coverage-guided Kernel Fuzzer. https://github.com/google/syzkaller. (2019).

[18] Bogdan Gribincea. 2009. Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all. (2009).

[19] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. Minneapolis, MN.

[20] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

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

[22] Dae R. Jeong, Kyungtae Kim, Basavesh Ammanaghatta Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[23] Dave Jones. 2018. Linux system call fuzzer. https://github.com/kernelslacker/trinity. (2018).

[24] Jan Kara. 2014. ext4: Forbid journal_async_commit in data=ordered mode. https://patchwork.ozlabs.org/patch/414750/. (2014).

[25] Kernel.org Bugzilla. 2018. Btrfs bug entries. https://bugzilla.kernel.org/buglist.cgi?component=btrfs. (2018).

[26] Kernel.org Bugzilla. 2018. ext4 bug entries. https://bugzilla.kernel.org/buglist.cgi?component=ext4. (2018).

[27] Michael Kerrisk. 2019. fsync, fdatasync - synchronize a file's in-core state with storage device. http://man7.org/linux/man-pages/man2/fdatasync.2.html. (2019).

[28] Eric Koskinen and Junfeng Yang. 2016. Reducing Crash Recoverability to Reachability. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL.

[29] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[30] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A Study of Linux File System Evolution. *Trans. Storage* 10, 1, Article 3 (Jan. 2014), 32 pages. https://doi.org/10.1145/2560012

[31] 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 ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[32] MITRE Corporation. 2009. CVE-2009-1235. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1235. (2009).

[33] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-consistency Bugs with Bounded Black-box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA.

[34] Ingo Molnar and Arjan van de Ven. 2019. Runtime locking correctness validator. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt. (2019).

[35] NCC Group. 2017. AFL/QEMU Fuzzing with Full-system Emulation. https://github.com/nccgroup/TriforceAFL. (2017).

[36] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[37] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China.

[38] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[39] Vijayan Prabhakaran, Lakshmi N Bairavasundaram, Nitin Agrawal, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2005. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. Brighton, UK.

[40] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux kernel library. In *Proceedings of the 9th Roedunet International Conference (RoEduNet)*. IEEE.

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[42] Red Hat Inc. 2018. Utilities for managing the XFS filesystem. https://git.kernel.org/pub/scm/fs/xfs/xfsprogs-dev.git. (2018).

[43] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[44] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. In *Proceedings of the ACM Transactions on Storage (TOS)*.

[45] Andrey Ryabinin. 2014. UBSan: run-time undefined behavior sanity checker. https://lwn.net/Articles/617364/. (2014).

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

[47] SGI, OSDL and Bull. 2018. Linux Test Project. https://github.com/linux-test-project/ltp. (2018).

[48] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[49] Silicon Graphics Inc. (SGI). 2018. (x)fstests is a filesystem testing suite. https://github.com/kdave/xfstests. (2018).

[50] Theodore Ts'o. 2018. Ext2/3/4 file system utilities. https://github.com/tytso/e2fsprogs. (2018).

[51] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[52] Junfeng Yang, Can Sar, and Dawson Engler. 2006. Explode: a Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA.

[53] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. 2006. Automatically Generating Malicious Disks Using Symbolic Execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[54] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2004. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Francisco, CA.

[55] Chao Yu. 2018. F2FS: f2fs_check_rb_tree_consistence. https://lore.kernel.org/patchwork/patch/953794/. (2018).

[56] Michal Zalewski. 2014. Bash bug: the other two RCEs, or how we chipped away at the original fix (CVE-2014-6277 and '78). https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html. (2014).

[57] Michal Zalewski. 2019. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl. (2019).

[58] Andreas Zeller, Holger Cleve, and Stephan Neuhaus. 2019. Delta Debugging: From Automated Testing to Automated Debugging. https://www.st.cs.uni-saarland.de/dd/. (2019).