Finding Bugs in File Systems with an Extensible Fuzzing Framework

SEULBAE KIM, MENG XU, SANIDHYA KASHYAP, JUNGYEON YOON, WEN XU, and TAESOO KIM, Georgia Institute of Technology

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. These bugs come in various flavors: buffer overflows to complicated 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 article, to highlight the potential of applying fuzzing to find any type of file system bugs in a generic way, we propose Hydra, an extensible fuzzing framework. Hydra provides building blocks for file system fuzzing, including input mutators, feedback engines, test executors, and bug post-processors. As a result, developers only need to focus on building the core logic for finding bugs of their 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 157 new bugs in Linux file systems, including three in verified file systems (FSCQ and Yxv6).

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

Additional Key Words and Phrases: File systems, fuzzing, bug finding, crash consistency

ACM Reference format:
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

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` [50], with 50K and 130K lines of code, respectively, witnessed 54 [29] and 113 [28] 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 [35] but also

This research was supported in part by NSF awards CNS-1563848, CNS-1704701, CRI-1629851, and CNS-1749711 ONR under grants N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware, and Google.
Authors' address: S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, Georgia Institute of Technology, 756 West Peachtree Street NW, Atlanta, GA 30308; emails: {seulbae, meng.xu, sanidhya, jungyeon, wen.xu, taesoo}@gatech.edu.

1553-3077/2020/05-ART10 $15.00
https://doi.org/10.1145/3391202

poses severe security threats [38, 58, 60]. 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` [48]) and tools (`fsck` [56]). However, these tools cannot handle the diverse types of 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. Although specialized checkers can often complement manually written cases in capturing various types of file system bugs [39, 49], 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, 47, 64]—has become the go-to approach, with thousands of vulnerabilities in real-world software, including critical ones [63], as its trophies. Without a doubt, file systems can be fuzzed, and existing OS fuzzers [20, 26, 41, 52] have demonstrated this viability by focusing either on mutating images as ordinary binary inputs [52], or generating random sets of file operation-specific system calls [20, 26, 41]. Unfortunately, they all fail to efficiently and comprehensively test file systems mainly for two reasons.

First of all, there are innate challenges in a fuzzing file system. Since a disk image is a large binary blob that can be fairly larger than the preferred size of the mutation target of general fuzzers, fuzzing throughput is critically degraded due to the heavy I/O involved in mutating images. In addition, generating context-aware workloads, such as considering the dependency that exists between an image and file operations executed on it, is challenging, and existing system call fuzzers [20, 26, 41] fail to emit meaningful sequences of file operations and cover deep code paths of a file system.

Second, even after addressing the first challenge, most fuzzers forgo the opportunity to find the dominant, diverse, and harder-to-catch category of file system bugs—semantic bugs [35]—because they only focus on memory safety bugs. Semantic bugs in file systems come in various flavors, including but not limited to violations of widely agreed properties (e.g., crash safety), nonconformance 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.

Hence, it is crucial to first address the challenges in fuzzing file systems efficiently, and broaden the spectrum of covered bug types to effectively and comprehensively find bugs in file systems. We build Hydra, an extensible fuzzing framework that is capable of discovering any type of file system bugs (in theory) in various file systems with full automation. As a framework, Hydra provides building blocks for file system fuzzing, including multi-dimensional input mutators, feedback engines, a test case 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 [49]) 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 article, we demonstrate how Hydra solves the prominent issues of existing file system fuzzers and then 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 Section 3.4.1), the SibylFS oracle [49], the existing file system–specific assertions inside the codebase, and the Kernel Address Sanitizer (KASan). As a result, Hydra found 123 memory errors, eight crash consistency bugs, four POSIX violations, and 23 logic bugs in popular and heavily tested Linux file systems including `ext4`, `Btrfs`, and `F2FS`, and even three crash consistency bugs in FSCQ and Yxv6, which have been proven to not have such bugs [10, 54].

Summary. This article makes the following contributions:

- To tackle diverse types of 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, 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. The implementation of Hydra is open sourced at https://github.com/sslab-gatech/hydra.
- Leveraging in-house developed and externally available bug checkers, Hydra has discovered 157 new bugs of four different types in various file systems, out of which 125 bugs have been acknowledged and 89 bugs have been fixed, which shows its worth.
- In this work, we extend the conference paper [31] to further discuss the motivation and design in greater detail, include more bugs found from another verified file system, Yxv6, and present in-depth analysis of bugs with test cases.

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 [35]. 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

Software bugs can be broadly categorized into semantic bugs, memory bugs, and concurrency bugs [36]. Memory bugs are caused by improper handling of memory objects. Concurrency bugs are caused in a multi-threaded environment by missing or erroneous synchronization of shared resources. Semantic bugs violate high-level rules or invariants, diverging from the programmer's intention and the original design. Bugs that are neither memory nor concurrency fall into the category of semantic bugs.

Table 1 summarizes three different types of semantic bugs and a memory safety bug that are often found in mainstream file systems, as well as related bug checkers that are specially designed for each bug type.

Crash inconsistency. 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/undeletable files. Therefore, crash consistency is a fundamental property relied upon by data-sensitive applications,

Table 1. List of Common Types of File System Bugs, Their Classes, Root Causes, and the Corresponding Checkers That Can Be Plugged into Hydra to Find These Bugs

Bug Type	Class	Description	Bug Checker
Crash Inconsistency	Semantic	Data not properly persisted upon system crash or power loss	SymC3 (Section 3.4.1)⋆, eXplode [59], B3 [39]
Specification violation	Semantic	Implementation not conforming to specifications (e.g., POSIX)	SibylFS [49]⋆, EnvyFS [22], Recon [13]
Logic bug	Semantic	Using wrong algorithms or making invalid assumptions	Built-in checks (e.g., [1])⋆
Memory error	Memory	Out-of-bound accesses, use-after-free, uninitialized read, etc	KASan [15]⋆, KMSan [16], UBSan [51]

⋆Bug checkers integrated with Hydra.

```
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 line 8, the expected size of y was 3000, but the size was 8000 after recovery)

Fig. 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.

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

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

Fig. 2. Specification non-conformance in ext4. `unlink()` returns an error code that is not compliant with POSIX but is acceptable to the Linux specification [49].

such as databases or servers. Unfortunately, there are very limited testing resources for this property apart from a handful of regression tests and the recent work: eXplode [59] and B3 [39]. eXplode, an *in situ* model checking approach, requires users to pay heavy implementation cost of modifying the file system code and writing system–specific checkers. B3, however, pursues an easy-to-use systematic testing by exhaustively generating test cases within bounds and concretely running them on the file system stack. With this design choice, B3 misses bugs that are beyond the bounded test space.

Specification violation. File system 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. As a counter-example, Figure 2 is a POSIX violation reported by Hydra, as the only allowed error code from the `unlink` syscall is `EPERM`, whereas the actual implementation returns `EISDIR`.[1] As in the example, if the file system does not conform with the specifications,

[1]Note that this does not violate the Linux specifications.

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

(failed to verify the extent tree refs)

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

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

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

Fig. 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; and when unmounting, a use-after-free is detected.

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 [49]. Unfortunately, The testing scope of SibylFS is limited to its synthesized test suite, which covers a small fraction of the entire test space.

Logic bug. 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* [62], 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, whereas existing file system test suites can rarely explore these corner states.

Memory error. Memory errors are common in file systems. Due to their high security impact, such as enabling remote code execution, several runtime checkers have been proposed to detect memory errors. The most prominent examples are the sanitizer series (i.e., KASan [15], KMSan [16], and UBSan [51]) 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

```
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)
```

Fig. 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.

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` [40]) to industrial tools (e.g., KT-San [18]) to research prototypes (e.g., SKI [12]). Although our current demonstration focuses on the other four bug types (see 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, 11]. Although these bugs can be systematically detected [45, 59, 61], the disk failures are beyond the scope of this article, as we assume that a persistent storage is reliable. Alternatively, Hydra tests the robustness of file systems by injecting corruptions in the file system images and checking whether file systems lets the corruption lead to unexpected bugs in the uppermost layer of the storage stack.

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` [53] and `xfstests` [55]) and testing tools (e.g., `fsck` [56]) 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 of 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 [39] in finding crash inconsistencies and SibylFS [49] 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—for instance, 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." This assumption is established

under their observation of the previously known crash consistency bugs that most of them are triggered by executing three or fewer *core* file operations. Given that, B3 enumerates all test cases that have one (seq-1) or two (seq-2) core operations, and a subset of seq-3 workloads. However, 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—for instance, more than a week if optimistic (Section 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 [9, 10, 24] and Yggdrasil (Yxv6) [54] 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 [39] as well), as well as two bugs in Yxv6, 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 [32], complementing the efforts of the formally verified file systems.

Motivations. Two components in a bug finding method affect the quantity and quality of bugs: (1) definition of a bug and corresponding core checking logic, and (2) the test case generator and the range of program states covered. 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 test cases contemplated by a 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 (Section 5.6).

2.3 Fuzzing as a Turnkey Solution

Fuzzing is a software testing method that repeatedly generates new inputs and injects them into a target program to trigger bugs. It is one of the most effective approaches in finding security bugs in modern software, as evidenced by the state-of-the-art fuzzer AFL [64] and its variants [3, 4, 14, 44]. Moreover, to apply fuzzing to the kernel, several frameworks [20, 41, 52] extend the AFL approaches to trigger kernel bugs by invoking randomly generated syscalls. 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 serves as the natural selection force in the evolutional process and 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.

2.4 Combining Efforts: Fuzzing Framework

This inspires us to design an extensible fuzzing framework that complements existing bug checkers with a fuzzing-based file system states explorer. Besides merely putting bug checkers and fuzzy input mutator together, our goal is to build a complete framework that provides a seamless workflow

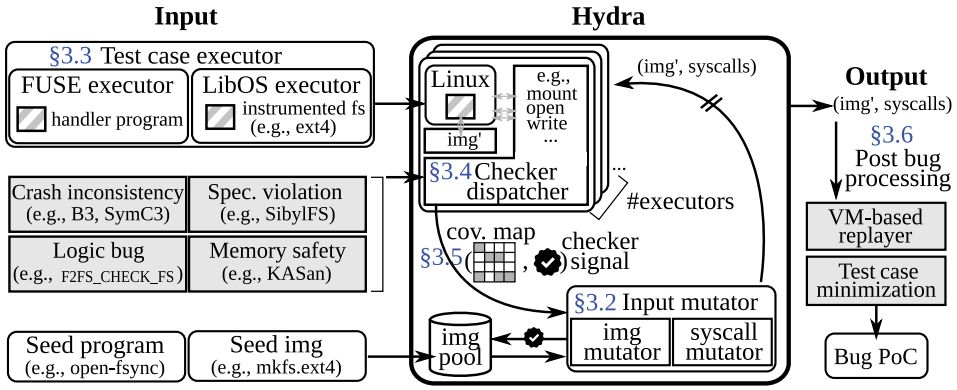

Fig. 6. Overview of Hydra's architecture and workflow.

of end-to-end bug detection, starting from input test case generation to buggy test case verification. In addition, with a framework, we can exploit the synergy of fuzzy input mutator and quality bug checkers. Instead of naively enumerating the test space, the input mutator *explores* the space to find test cases that are likely to trigger bugs based on the guidance (i.e., feedback) given by a precise checker of a certain type of bugs. In addition, further experience in fuzzing OS kernels has shown unique challenges, such as reproducing bugs and creating proof-of-concepts (PoC) (i.e., reproducible test cases) of reasonable sizes. 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 reuse 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

In the face of diverse file system bugs, we propose to build a comprehensive framework—Hydra [31]—to tackle the challenges of applying fuzzing to file system, and to complement existing and future bug checkers by providing a set of commonly required components, such as input mutator, feedback engines, test case executors, and bug post-processors, 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 is a pack of both a file system image to be mounted and a sequence of syscalls to be executed on the mounted image. The input mutator subsequently mutates either the image or the syscalls or both, and produces a batch of test cases (Section 3.2). The test cases are sent to a test case executor that always starts in a clean-slate state, mounts the given image, and executes the syscalls (Section 3.3). The visited code paths are profiled into a bitmap by the coverage tracker instrumented when compiling the target file system. Meanwhile, the bug-checking dispatcher (Section 3.4) invokes the necessary checks, such as runtime assertions or an out-of-band emulation (Section 3.4.1, Section 3.4.2). 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 (Section 3.5). If

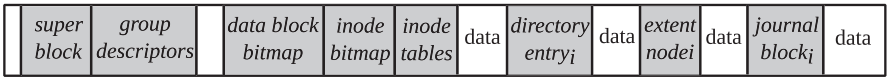

Fig. 7. The on-disk layout of an ext4 image. The gray blocks show metadata in use, which occupies merely 1% of the image size. Some of them, including extent tree nodes, directory entries, and journal blocks, are scattered in the image, whereas others (superblock, group descriptors, etc.) are located in the beginning.

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 to create a simplified test case for the ease of analyzing and fixing the bug (Section 3.6).

3.2 Input Mutator

The input space of a file system consists of two major components: a file system image to mount, and file operations that access, read from, and write to the mounted image. Thus, to trigger a file system bug,[2] one needs to (1) mount a corrupted file system image or (2) execute a crafted sequence of syscalls to access/modify the data/metadata stored on the mounted image, or most likely, (3) combine both steps. Therefore, unlike traditional program fuzzers that only need to mutate user input [19, 33, 64], or OS fuzzers that mutate syscall sequences without consulting the dependency with the mounted image [20, 26, 41], a fuzzer tailored to file system needs to synthesize both aspects when generating meaningful test cases. Considering this, Hydra exploits the synergy between mutating both file system images and syscall traces.

Image mutation. A file system disk image can be corrupted for many reasons. The storage media may fail due to electrical or mechanical issues, or the disk firmware or device drivers may be buggy, resulting in an inconsistent disk image. In response to such failure, file systems implement their own mechanisms for error checking and recovery. Even though their philosophies and corresponding approaches may differ, a robust file system should be capable of handling such corruption and reacting gracefully to failures to prevent unexpected errors. This is the aspect we aim to test through the image mutation.

A naive approach would be to treat the whole image as a black box and mutate bytes at random offsets in the image. However, this is neither practical nor efficient. Hydra's approach leverages the fact that a file system image is highly structured into user data chunks (e.g., file content) and a few management structures called *metadata*, as shown in Figure 7. For mounting and executing a majority of file operations, only metadata is consumed, which constitutes merely 1% of the image size, making it a feasible target for random mutation. Using these facts, Hydra first maps the entire image into a memory buffer. Then it scans the image to locate all metadata blobs. As the image structure differs across file systems, a file system–specific image parser is required. Utilities and libraries provided by file systems (e.g., `e2fsprogs` of `ext4`) can be utilized for developing these parsers. Once the metadata blobs are identified, it applies several common mutation strategies [64] (bit flipping, arithmetic operation on random bytes, etc.) to randomly mutate the bytes of the metadata as described in Figure 8.

After mutating the entire metadata blob, Hydra reassembles each metadata block back to its corresponding position inside the memory buffer, which stores the original full-size image. As a result, Hydra obtains a corrupt disk image that partially reflects the consequences of various disk-failure scenarios. Here, to focus on testing the corner cases that are likely to trigger unexpected bugs file systems fail to handle, Hydra recalculates the checksum values of all metadata blocks,

[2]We refer to the unexpected errors that file systems fail to handle. For example, if `mount` syscall fails and returns a proper error code, that is not a bug. However, if the kernel crashes after `mount` succeeds, that is considered a bug.

```
# Class ImageMutator
def mutate_image(meta_buffer):
    choice = Random.randint(0, 8)
    if choice == 0:
        return flip_bit_at_random_offset(meta_buffer)
    elif choice == 1:
        return set_interesting_byte_at_random_offset(meta_buffer)
    elif choice == 2:
        return set_interesting_word_at_random_offset(meta_buffer)
    elif choice == 3:
        return set_interesting_dword_at_random_offset(meta_buffer)
    elif choice == 4:
        return inc_random_byte_at_random_offset(meta_buffer)
    elif choice == 5:
        return inc_random_word_at_random_offset(meta_buffer)
    elif choice == 6:
        return inc_random_dword_at_random_offset(meta_buffer)
    else:
        return set_random_byte_at_random_offset(meta_buffer)
```

Fig. 8. Pseudo-code of how HYDRA randomly mutates metadata blocks.

```
# Class Hydra
def release_image(self, meta_buffer):
    pos = 0
    for meta_block in self.meta_blocks:
        meta_block_buffer = meta_buffer[pos:pos + meta_block.size]
        if meta_block.chksum_offset is not None:
            self.fix_chksum(meta_block_buffer, meta_block.chksum_offset)
        copy_buffer(self.image_buffer[meta_block.offset],
                meta_block_buffer, meta_block.size)
        pos += meta_block.size
```

Fig. 9. Pseudo-code of how HYDRA releases the mutated metadata blocks back to a full-size image for testing.

by following the specific algorithm adopted by the target file system, and fills the value at the recorded offset of the checksum field, as shown in Figure 9. This ensures that the tested mutated images are corrupt but mostly valid (i.e., not being early rejected by sanity checks) so that they enable HYDRA to reach deep code paths.

Syscall mutation. The goal of syscall mutation is to generate diverse, complex, and, more importantly, image-aware file operations, which are likely to trigger deeper, possibly un-tested code path of the file system when executed in a sequence. Similar to existing OS fuzzers [20, 26], HYDRA mutates syscall sequences in two ways: (1) argument mutation (randomly selecting one of the existing syscalls in the sequence and mutating its argument(s)) and (2) syscall generation (appending a new randomly chosen syscall to the end of the sequence) (Figure 10).

To generate mostly valid syscalls that explore deep into file system logic, instead of being early rejected by an error checking routine, HYDRA leverages the semantics of the following syscall argument types:

- —For *flags* with a clearly defined set of possible values, HYDRA selects and combines these values randomly (e.g., `int flags` for `open()`).
- —For *size-like integers*, HYDRA generates random numbers within a certain range and also tries sensitive values (e.g., page size or block size).

```
# Class SyscallMutator
def mutate_syscall(self):
    new_program = Program(self.program)
    syscall_index = Random.randint(0, len(self.program.syscalls))
    syscall = self.program.syscalls[syscall_index]
    args = [i for i in range(len(syscall.args)) \
            if not may_effect_status(syscall, i)]
    arg = Random.choice(args)
    mutated_arg = self.generate_arg_by_status(syscall, arg)
    new_program.syscalls[syscall_index].args[arg_index] = mutated_arg
    return new_program

def generate_syscall(self, sysno=None, args=[]):
    new_program = Program(self.program)
    new_status = Status(self.status)
    syscall = Syscall()
    if sysno is None:
        syscall.sysno = Random.choice(FS_SYSNOS)
    else:
        syscall.sysno = sysno
    for arg in args:
        syscall.add_arg(arg)
    for i in range(len(args), SYSCALL_ARG_NUM[syscall.sysno]):
        syscall.add_arg(self.generate_arg_by_status(syscall, i))
    new_program.add_syscall(syscall)
    new_status.update(syscall)
    if self.test_bug_type == "consistency" and syscall.sysno != sysno_of_fsync:
        (new_program, new_status) = self.generate_syscall(sysno=sysno_of_fsync)
    return (new_program, new_status)
```

Fig. 10. Pseudo-code of how HYDRA randomly mutates existing system calls and generates new ones given a program.

- For *buffer objects* that are used to communicate bulk information with the kernel, HYDRA uses pre-allocated buffers filled with random data.
- For *file descriptors*, HYDRA maintains the list of open file descriptors in the runtime and randomly picks one if required by the mutation.
- For *paths*, HYDRA maintains the list of available and stale paths on the mounted image and randomly picks one if required by the mutation. If the path is used to create a new file or directory, HYDRA randomly generates a valid new path that is located under an existing directory.
- For *extended attributes*, HYDRA randomly picks a recorded attribute name available in the file system (e.g., `user.mime_type` or `system.posix_acl_access`).

For a newly generated syscall, HYDRA appends it to the program and summarizes the potential changes to the file system caused by the syscall and updates the speculated status of the image correspondingly. For instance, `open()`, `mkdir()`, `link()`, or `symlink()` may create a new file or directory, whereas `open()` also introduces an active file descriptor; `rmdir()` or `unlink()` removes a file or a directory from the image; `rename()` updates the path of a file; and `setxattr()` or `removexattr()` updates a particular extended attribute.

Exploiting the synergy. To fuzz file system 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—in other words, no new code path is discovered—HYDRA then invokes the syscall mutator to alter arguments of the existing syscalls. If still no

interesting test cases are found after certain rounds, Hydra eventually generates and appends new syscalls.

Scheduling image and syscall mutation in such an order is effective for the following two reasons. First, conceptually, the metadata represents the initial image state, whose impact on the executions of file operations gradually decreases when the state of the image is altered by syscalls. Hence, Hydra always tries to mutate metadata first. Second, introducing new file operations exponentially increases the mutation space for later rounds and may also offset the changes from past operations of the image. Therefore, Hydra prioritizes mutating existing syscalls over generating new ones.

Assisting bug checkers. On top of the generic strategy, Hydra further assists bug checkers by adopting checker-specific strategies. For example, when generating test cases for crash consistency testing, a valid `fsync` syscall is appended to generate a persistence point, as shown in lines 27 and 28 of Figure 10.

3.3 Test Case Executor

Hydra's test case executor is the module where the generated test cases are concretely executed on the targeted file system. In general, 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 the checker dispatcher (Section 3.4), which calls a checker, collects results, and then provides an additional dimension of feedbacks to the feedback engine (Section 3.5). Hydra supports both in-kernel file systems (e.g., `ext4`), and FUSE (Filesystem in Userspace) file systems (e.g., FSCQ), and as their properties differ greatly, two different types of executors are provided: (1) a library OS-based executor for supporting in-kernel file systems and (2) a FUSE-based executor for supporting those that run in userspace.

Library OS-based executor. For the sake of performance, the traditional OS fuzzers [20, 41, 52] reuse virtualized instances (KVM, QEMU, etc.) to run a target OS without reloading a fresh copy of the kernel or file system image for fuzzing different test cases. Because of the accumulated non-deterministic OS states, this approach, unfortunately, impedes the stable PoC generation, which developers can reproduce and debug [17]. Although rejuvenating OS solves this issue, it can be as slow as a couple of seconds. Hence, Hydra utilizes a library OS-based executor, which incurs negligible time (tens of milliseconds) to run kernel and file system logic, and 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.

FUSE-based executor. Unlike in-kernel file systems, a typical FUSE-based file system is implemented as a stand-alone application, a so-called handler program, which includes a top-level interface that handles requests (e.g., mount operation) through the `libfuse` library. The FUSE-based executor launches the registered handler program of the file system and lets it mount the given image. Once the executor detects the mount, it then executes the syscalls and consults the checker dispatcher to check for the existence of bugs.

Since both (i.e., library OS- and FUSE-based) executors assure that each test case is executed on a clean-slate internal state, Hydra guarantees a high reproducibility of detected bugs, without having to compromise the performance, because the executors can be relaunched with negligible overhead.

3.4 Checker Dispatcher

Invoked by the test case executor, the checker dispatcher launches a checker plugin that corresponds to the targeted bug type. In theory, any type of bug checker can be plugged into Hydra, but the checkers may have different interfaces. For example, SibylFS, a POSIX compliance checker,

is available as a stand-alone binary, which takes as input a specifically formatted test case that is different from what Hydra's input mutator generates. Meanwhile, another example, KASan, is integrated into the kernel and can be enabled by configuring a flag when compiling the kernel. Hydra's checker dispatcher takes these differences into consideration and provides a unified interface, which abstracts away how a test case is consumed by different checkers, and how test results are transferred from them. In the end, from the framework's perspective, the checker dispatcher takes the test case as is and returns a binary result: interesting (bug) or not interesting (no bug). In addition, to make the integration of new checkers to Hydra as straightforward as possible, an API is provided, which helps developers write Hydra-compatible checkers. The following sections discuss how our in-house-developed checker SymC3 and other checkers are plugged in to expose various types of file system bugs.

3.4.1 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 the 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 [21, 27].

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, such as names, type, size, link count, and attributes, as well as type-specific properties like 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 (lines 2 and 8, respectively), whereas `fdatasync` only commits data and those metadata that are related to data (e.g., size, checksum) to disk (line 5). Other syscalls (lines 4 and 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.

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

#	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, 0.86	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

Boxed region represents the snapshots taken and strikethrough text maintains the history of data and metadata changes before synced.

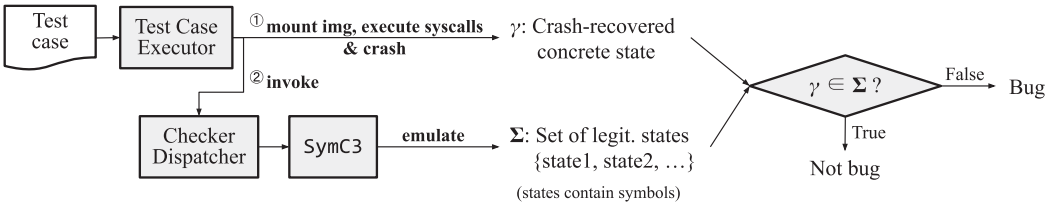

Fig. 11. Overview of how Hydra utilizes SymC3 for detecting crash consistency bugs. ①and ②are pipelined.

Enumerating crash states. At any stage, SymC3 is capable of generating the set of legitimate 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 (states S1, S2, and S3).

Checking crash consistency. For crash consistency testing only, Hydra appends a persistence operation (i.e., `fsync`) to the generated syscall sequence of the test case (see Figure 10). The process of checking crash consistency is illustrated in Figure 11. Given a test case, the test case executor (Section 3.3) pipelines the crash consistency testing: (1) it invokes SymC3 for the crash state emulation, and (2) it simulates a crash and a recovery process by mounting the given image, concretely executing syscalls, and remounting the copy of the mounted image at a separate mount point right after executing the last syscall (`fsync`). SymC3 compares this crash-recovered image with the emulated legitimate states. If SymC3 finds that the image does not match any of the legitimate 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 8,000, which is not one of the legitimate states.

The example 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, whereas `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 Section 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 the 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).

3.4.2 Other Checker Plugins. Besides the homegrown 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 [49] 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 each syscall in an observed trace yields correct return value as POSIX specification defines, given the initial image and the sequence of syscalls. To bridge Hydra with SibylFS, we run the SibylFS oracle in a stand-alone 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 Section 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, such as `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. Hydra also looks for memory errors leveraging KASan [15], especially out-of-bound and use-after-free bugs in file system implementations. Specifically, KASan tracks kernel object life cycles by hooking the SLAB and SLUB memory allocators and further uses compile-time instrumentations to insert sanity checks before every memory access. To integrate KASan with Hydra, we enabled the `CONFIG_KASAN=y` option and additionally allow KASan to instrument the target file system for memory access sanity checks. 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.

3.5 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 and checker-defined signal.

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 buggy, 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 toward the yet-to-be-tested part of the specification.

3.6 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 [17]. 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 bug PoC, which is a reproducible test case.

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., the Linux kernel library (LKL) [46]) 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 bug PoC should be minimal—that is, 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 [65]. 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 syscall 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 [23, 42].

3.7 Summary

In a nutshell, Hydra, as a full-fledged framework, provides an automated process of revealing, in theory, any *developer-defined buggy situation*. It represents an important design point that cleanly separates the concerns of developers: Hydra takes charge of automated input exploration, checker incorporation, and validation of found bugs on behalf of developers, whereas the developers can solely focus on writing a reliable checker for a bug. Such separation of concerns drastically improves the quality of bug finding in terms of both accuracy and efficiency (Section 5).

4 IMPLEMENTATION

Hydra adopts the basic fuzzing infrastructure from AFL 2.52b [64], including the fork server, code coverage bitmap, and test case scheduling, but replaces a few key components, including

Table 3. Implementation Complexity of Hydra and SymC3, and of Integrating SibylFS, KASan, and File System–Specific Built-In Checks

Component	LoC	Language
Hydra Framework		
Input mutators	8,507	C++/Python
LKL-based executor	2,190	C++
FUSE-based executor	176	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	3,585	Python
`ext4` extension	3	Python
`F2FS` extension	3	Python
`Btrfs` extension	35	Python
FSCQ extension	4	Python
Yxv6 extension	19	Python
SibylFS Integration		
Test case parser	188	Python
Converter	271	Python
Logger and extra	138	Python
KASan Integration		
Kconfig change	1	—
File System Built-in Checks		
Kconfig change (total)	7	—
`ext4`	2	—
`F2FS`	1	—
`Btrfs`	4	—

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—that is, 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 LKL [46] as the library OS for the library OS-based executor. The official LKL is based on Linux kernel v4.16, and we ported it to v5.0. When compiling the 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. The 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.

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).

```
// MutationStage.cpp
KnownSyscalls::KnownSyscalls(){
    add(SYS_read);
    add(SYS_write);
    add(SYS_open);
    add(SYS_rename);
    // add(SYS_gendents64);
    // ...
}
```

Fig. 12. A snippet from Hydra's input mutator code, where developers register syscalls to generate and mutate. With this configuration, Hydra will generate test cases tcomprise only `read`, `write`, open, and `rename` syscalls.

```
@type script
# creating a file without providing a mode is
# unspecified in Posix. For Linux, the
# default mode is 0o101
open "f2.txt" [O_CREAT;O_WRONLY]
close (FD 3)
stat "f2.txt"
```

Fig. 13. `file_descriptors/adhoc_open_creat_no_mode-int.trace` from SibylFS's test suite.

Customizing test case generation. By default, Hydra generates 27 system calls[3] that are specific to file systems. When generating test cases for testing memory safety bugs, all of them are utilized. However, there are cases where developers can utilize the domain-specific knowledge and focus on generating/mutating a smaller set of syscalls to increase the chance to hit bugs. For example, syscalls that affect the metadata, such as `chmod`, are more likely to trigger crash consistency bugs than syscalls that simply read metadata, such as `stat`. The input mutator of Hydra handles this by making the list of syscalls configurable; developers can place restrictions on the syscall sequences by simply adding the syscall numbers they want to have in the test cases, as shown in Figure 12.

Integrating existing checkers. The modularized structure of Hydra allows developers to readily integrate existing checkers with a minimum implementation effort. As described in Section 3.3, the test case executor receives the generated test case, executes it, and dispatches a checker. In case of the in-kernel checkers, such as KASan for memory safety bugs and file system–specific checks for logic bugs, there is no implementation cost, because these checkers can be enabled when compiling LKL and are automatically invoked by LKL while executing test cases. For external checkers, a developer has to convert the Hydra-style test case into the form that the utilized checker favors. For example, the SibylFS oracle requires an input to be formatted as a custom trace file, as shown in Figure 13. We wrote a test case parser and converter to convert a Hydra's test case, which consists of a file system image and a full-fledged C code, into a trace that is compatible with SibylFS. Overall, including the code for a logging functionality, we wrote less than 600 lines of Python code (see Table 3) for a seamless integration of SibylFS. Please note that SibylFS is written in more than 10,000 lines of OCaml code.

Writing a new checker. The minimum requirement for writing a compatible checker is that it needs to take a Hydra test case as input and return the test result as a binary value. On top of that, developers can optionally provide supplementary diagnosis results, such as an error message and

[3] `open`, `read`, `pread64`, `write`, `pwrite64`, `rename`, `mkdir`, `rmdir`, `link`, `symlink`, `unlink`, `readlink`, `chmod`, `stat`, `lstat`, `lseek`, `access`, `getdents64`, `fallocate`, `truncate`, `ftruncate`, `setxattr`, `listxattr`, `removexattr`, `utimes`, `fsync`, and `fdatasync`.

logs to assist further analysis of bugs. If the requirement is met, the rest, including the checking logic, can remain completely black box to the Hydra framework.

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 (Section 5.1). The effectiveness can be explained from three aspects: (1) a high fuzzing speed allows Hydra to explore file system states quickly (Section 5.2), (2) the dual-aspect input mutation allows Hydra to explore more execution paths than both existing OS fuzzers and specialized bug checkers (Section 5.3), and (3) the addition of checker feedbacks allows Hydra to further lean its fuzzing effort toward checker-defined states besides greedy path exploration (Section 5.4). Beyond hitting more bugs, we further evaluate Hydra's performance in bug reproducibility and test case minimization (Section 5.5). Evaluations on the in-house developed crash consistency checker are performed to support how it fares against prior works (Section 5.6), and an additional evaluation on the merit of having a comprehensive fuzzing framework besides regression tests is performed by analyzing the lifespan of the bugs Hydra detected (Section 5.7).

Experimental setup. We run Hydra on a 2-socket, 24-core machine running Ubuntu 16.04 with Intel Xeon E5-2670 and 256 GB of RAM. We tested `ext4`, `Btrfs`, `F2FS`, and `XFS` in Linux kernel v5.0, and two verified file systems: FSCQ (`sosp17` branch) and Yxv6. We also tested `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 (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 2 GB of RAM.

5.1 Bug Hunting in Popular File Systems

Across intermittent runs during a 10-month period of development, Hydra discovered 157 new bugs in total, of which 125 have been confirmed and 89 bugs fixed (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, 34 are semantic bugs that do not cause kernel panics when triggered and hence cannot be found by prior OS fuzzers. Even though the memory bugs found seem to outnumber the semantic bugs, in fact, the ability of Hydra to find rare, hard-to-detect semantic bugs, such as crash consistency bugs in verified file systems, is invaluable. This sheds light on how to stress not only memory errors but also the semantic parts in file systems by integrating specialized checkers into the Hydra fuzzing framework. Please note that even though these checker plugins existed before Hydra, they did not reveal any of the found bugs without being assisted by Hydra's input space exploration.

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 14, and we make the following 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 (seven logic bugs and 10 memory errors) can be triggered by merely mounting a corrupted image, which shows the importance of image mutations. Please note that we do not count a mount failure as a bug, because such a case shows that a file system is properly handling a corrupt image

Table 4. Hydra Found 157 New Bugs Along with Four POSIX Violations

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	1 w	19/22/19	4w
Btrfs	1/4/1	8w	7/7/3	1w	2	1w	30/30/20	1w
F2FS	2/3*/2*	4w	16/16/16	1w	1	1w	19/19/19	1w
XFS	0/0/0	—	0/0/0	1w	4	1w	11/17/7	1w
GFS2	—/—/—	—	—/—/—	—	—	—	0/14/0	1w
HFS+	—/—/—	—	—/—/—	—	—	—	7/8/1	1w
ReiserFS	—/—/—	—	—/—/—	—	—	—	8/13/0	1w
VFAT	—/—/—	—	—/—/—	—	—	—	0/0/0	1w
FSCQ	1/1/1	1w	—/—/—	—	—	—	—/—/—	—
Yxv6	2/2/0	1w	—/—/—	—	—	—	—/—/—	—
Total	7/11*/4*	18w	23/23/19	3w	4	3w	94/123/66	11w

A: **A**cknowledged, **R**: **R**eported, **F**: **F**ixed, **T**: **T**ested time (in weeks).
Hydra successfully reproduced all cases except four on VM. 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. Tested time (T) includes the time spent for developing (or integrating) and debugging each checker. For example, it required a total of 8 weeks for us to develop SymC3 and run Hydra with it to find four bugs in Btrfs.

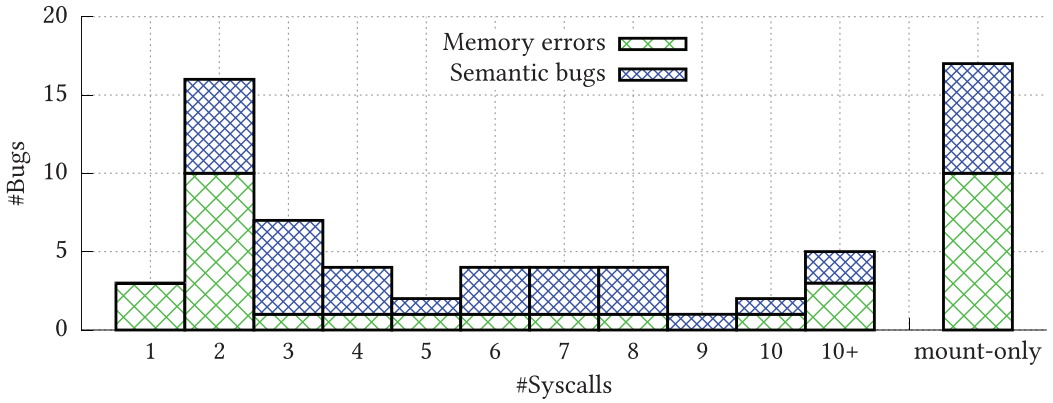

Fig. 14. The number of syscalls required to trigger each file system bug found by Hydra after minimization.

by refusing to mount. We only consider the cases that triggers bugs after the mutated image is successfully mounted (i.e., mount returns zero) as mount-only bugs.

Bug complexity. A file system bug can be triggered by as little as one syscall[4] 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 of the bugs that require file operations to manifest, 57% of memory errors can be triggered with just one or two syscalls, whereas 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

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

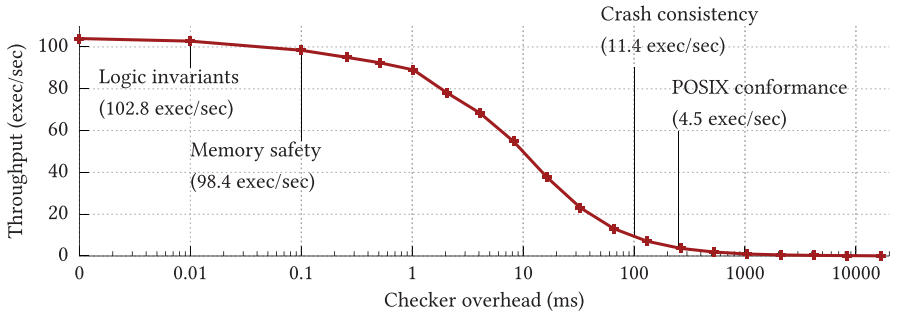

Fig. 15. Performance of Hydra's state exploration with checkers exhibiting different overheads, indicating the expected throughput when a developer integrates a custom checker with Hydra.

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 15). 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. However, 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 book-keeping (e.g., updating the AFL coverage bitmaps).

For comparison, a VM-based approach takes at least 1.4 seconds (0.7 exec/sec) to achieve the same effect—that is, a clean-slate kernel and file system for every test case, which is 100× slower than our libOS-based executor. In addition, although the throughput might be low for expensive checkers, this may be compensated by paralleling Hydra, similar to distributed fuzzers like syzbot [17].

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 16. 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.

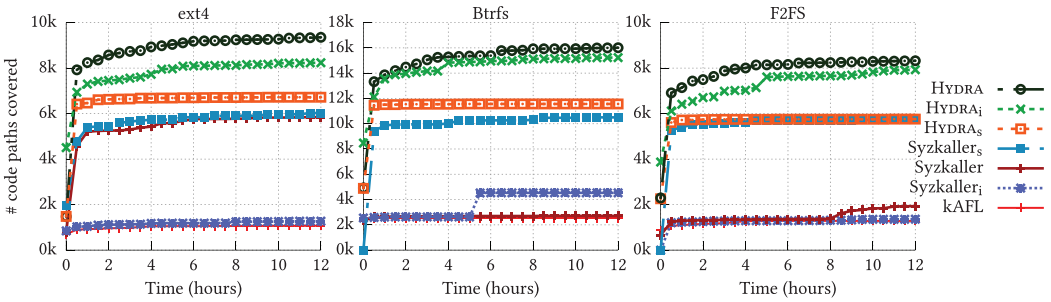

Fig. 16. Code coverage for 12 hours of fuzz testing with kAFL, Syzkaller, HYDRA, and their variants (*i* for image-only and *s* for syscall-only mutation). It indicates that HYDRA aggressively explores the input space that both kAFL and Syzkaller could not reach, and combining both image and input mutators exhibit a synergistic effect in exploring different parts of the input space.

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, whereas 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 and 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 Section 3.2), whereas 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 17.

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 2,500, and only two directories and two files are used as path arguments. However, HYDRA randomly mutates these arguments, discovering more code paths.

SibylFS test suite. Similarly, HYDRA yields more coverage than the test suite synthesized 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, although 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—that is, the states in which the

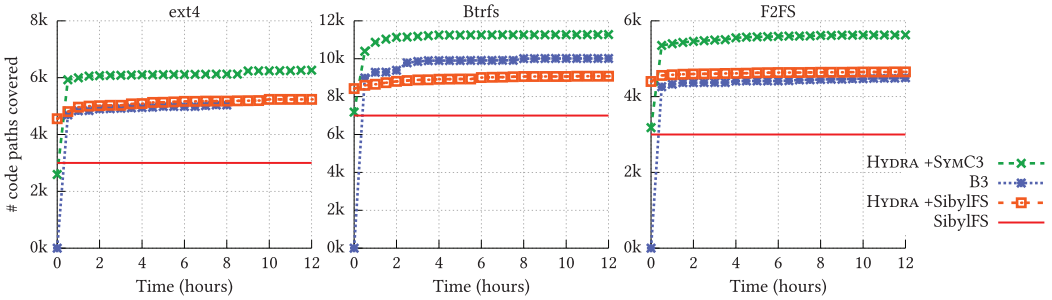

Fig. 17. Code coverage for 12 hours of checking crash consistency bugs by using a static mutator, B3, and our feedback-driven fuzzer, HYDRA with SYMC3, as well as checking specification violation using SibylFS's test suite versus HYDRA's dynamically generated test cases. In finding both bug types, HYDRA not only visits 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 left-most graph), and SibylFS's test suite completes within an hour (the straight red lines).

Table 5. Effectiveness of Checker Feedbacks

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

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. Test synthesizers tightly coupled for a specific bug type may hard code this intention in their test generation algorithms that try their best to enforce that the generated tests trigger the "favored" states. 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[5] and collect all 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 rerun these seed cases with checker enabled again and see how many unique bugs are found. The results are shown in Table 5.

[5]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.

Table 6. Number of System Calls Used to Trigger New Bugs Before and After Minimization

#	FS	#Syscall (min.)	B3 Sequence	Required Bound Relaxation
1	ext4	36→ 3 (91.7%)	seq-1⋆	Requires support for chmod
2	Btrfs	164→ 12 (92.7%)	seq-5	Requires longer sequence
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	Requires longer sequence
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	(Not reported by B3)
10	Yxv6	6→ 2 (66.6%)	seq-2	(Not tested by B3)
11	Yxv6	6→ 3 (50.0%)	seq-2	(Not tested by B3)

Even after minimization, all test cases, except FSCQ and Yxv6 bugs, cannot be reached by B3's input generator. The star (⋆) indicates that some relaxation of boundaries is required. For example, bug#3 requires one core operation (i.e., chmod) but needs to open and persist an arbitrary (seemingly irrelevant) file. This is not a case reachable by B3's seq-1 workloads.

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 and test case minimization.

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—for instance, (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). Only four bugs (one logic bug and three memory bugs) are not always replayable, although they are acknowledged by the developers, and the reason is that the 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 the 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.

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 bug PoC from 69.5 to 5.8, yielding a 91.6% reduction. As a snippet of the effectiveness of the minimizer, Table 6 shows how it reduces the syscalls in the bug PoC for the 11 crash consistency bugs Hydra found.

Table 7. Comparing Hydra with B3 in Terms of Execution Time and Precision in Testing Btrfs of Linux v4.16

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

T-Gen: time to generate, T-Comp: time to compile.
Hydra generates test cases on the fly, whereas B3 requires extensive resources for preprocessing when testing seq-1/2 test cases, Hydra incurs no false positives, whereas B3 reports 31K incorrect consistency errors.

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 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* of 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,[6] (3) not supporting crucial file system operations (e.g., `chmod`) or (4) not supporting different file types (e.g., `FIFO` file). The last column of Table 6 shows the specific bound that needs to be relaxed from B3's input generator to generate the corresponding buggy test case.

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 Section 3.4.1, the oracle is *one* of the possible post-crash states, and B3 ends up flagging legitimate cases as bugs. For example, if a file is resized multiple times but not persisted before a crash, the file in the recovered image can be of any size that it was once resized to. However, B3 only considers the final size (i.e., the size right before the crash) as a correct metadata and falsely identifies all other correct states as bugs, in consequence. Because of this systematic limitation, B3 reported 31K incorrect consistency errors from the `Btrfs` file system in the v4.16 kernel, whereas 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 5 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. However, because of fuzzing, the input generation in Hydra is dynamic and requires no preprocessing. In addition, the speed of execution

[6]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.

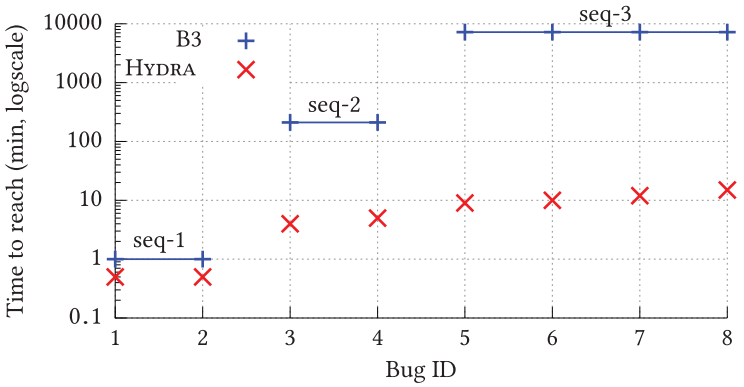

Fig. 18. Time to reach the eight crash consistency bugs in Btrfs, found and reported by B3. Bugs 1 and 2 require seq-1, bugs 3 and 4 require seq-2, and bugs 5 through 8 require seq-3 workloads, respectively, to be triggered.

of Hydra is orders of magnitude faster than that of B3, further showing Hydra's usability as a large-scale framework.

Time to reach bugs. B3 exhaustively tests all execution sequences in a limited test space, whereas Hydra uses a randomized approach to explore a larger test space. To evaluate the efficacy of the feedback-driven input generation of Hydra over exhaustive search, we compared the time to reach bugs that both Hydra and B3 found. As B3 fails to generate test cases for all 11 new bugs that Hydra found, we chose to use 8 Btrfs bugs that B3 previously detected for comparison. Among the eight cases, two are seq-1, two are seq-2, and the remaining four are seq-3 workloads. For a fair comparison, we modified the input mutator of Hydra to (1) not generate test cases that are longer than B3's seq-3 workloads and (2) select arguments from the exact set of values that B3's workloads use. The result is shown in Figure 18. Hydra successfully found 8 bugs; it reached two seq-1 workloads in 30 seconds and the longest workload after 15 minutes of execution. Meanwhile, as shown in Table 7, B3 required orders of magnitude longer time to exhaustively generate all seq-1 to -3 workloads. This proves that Hydra's feedback-driven input generation leads to the efficient exploration of a vast input space to find buggy cases without having to look through all test cases.

5.7 Lifespan of Detected Bugs

To analyze and compare the effectiveness of Hydra against that of existing regression testing approaches, we measure the lifespan of crash consistency bugs that Hydra detected from in-kernel file systems: ext4, Btrfs, and F2FS. The PoC of each bug is tested against all stable kernels starting from v4.1 to v5.3, which currently is the latest stable kernel.

The result is shown in Figure 19. Typically, regression testing is carried out to ensure that software works correctly after new pieces of code are added. Even though Linux kernel developers put a lot of effort into testing the prepatch (i.e., release candidate, or RC) kernels before releasing the stable version, we can still find that more than half of the detected bugs (bugs 2, 4, and 6–8) are introduced at some point and have been present in several stable versions until being detected by Hydra. This shows that Hydra is an effective approach to complement regression tests that completely missed these bugs. Among these bugs, three cases were fixed in the v5.1 release after developers took actions in response to our bug reports, and one (bug 8) was patched before we reported. The other four bugs are yet to be fixed and still remain in the latest stable kernel. Further analysis of these bugs will be given in Section 6.

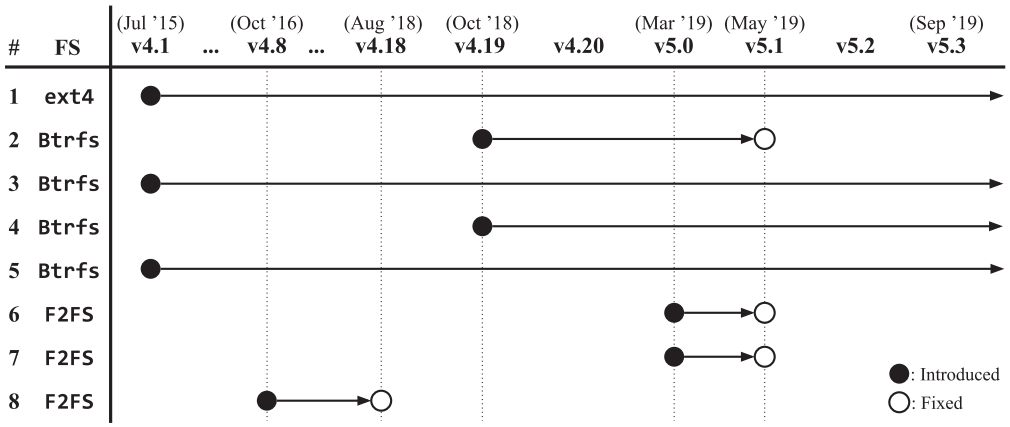

Fig. 19. Lifespan of crash consistency bugs in stable Linux kernel, from v4.1 to v5.3. The black dot (●) denotes the version in which bug is introduced, the white dot (○) marks the version in which bug is fixed, and the arrow (→) refers to the affected versions. Note that the bug ids (#) are consistent with those of Table 6.

```
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
```

Fig. 20. FSCQ: `fsync` fails to persist directory `A` after `fsync` on root directory. After a crash and recovery, only `foo` is in the file system.

6 CASE STUDY

6.1 Crash Consistency Bugs in Verified File Systems

This section breaks down the crash consistency bugs found in FSCQ and Yxv6, which are verified file systems.

FSCQ. FSCQ [9] 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 20 shows the bug PoC when `fsync` fails to persist a directory entry. This bug triggers an uncovered part in their proof, as noted by the developer [7]:

> "[T]he 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 of 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.

```
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
```

Fig. 21. FSCQ: fdatasync fails to persist data in file foo. After a crash and recovery, foo is empty.

```
1 mkdir("./A");
2 mkdir("./A/B");
3 fd_foo = open("./A/B/foo", O_CREAT | O_RDWR, 0777);
4 sync(); // persist three new inodes, A, B, and foo persist
5 mkdir("./C");
6 rename("./C", "./A"); // should fail if POSIX-compliant
7 sync();
```

Fig. 22. Yxv6: Children inodes A/B and A/B/foo are orphaned when renaming an empty directory C to A. After a crash and recovery, directories A, B, and file foo are lost.

```
1 char buf[2048] = {0, };
2 fd_foo = open("./foo", O_CREAT | O_RDWR, 0777);
3 write(fd_foo, buf, 1947);
4 int ret = ftruncate(fd_foo, 2791); // ret is 0 (success code)
5 fsync(fd_foo); // size of foo should be 2791
```

Fig. 23. Yxv6: File size is not properly adjusted by ftruncate. After a crash, the size of foo is 1,947.

Figure 21 shows the bug PoC when fdatasync is not persisting data written to files. According to FSCQ developers [8], this is due to different interpretations of the fdatasync on the Linux man page [30], 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 4,000 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.

Yxv6. Yxv6 [54] is yet another verified file system, in which possible disk states after a crash are defined as a subset of those allowed by the specification. Hydra found two interesting bugs in the latest GitHub release of Yxv6 (commit e1de61), which eventually lead to crash inconsistency.

Figure 22 is one buggy test case, where persisted inodes are lost in face of a crash. The PoC creates directories A and A/B, and a regular file foo under A/B/, then persists them altogether through sync (lines 1–4). Then a new empty directory C is created and renamed to A (lines 5 and 6). Upon execution of rename, directories A, A/B, and the file A/B/foo are unlinked, and the image ends up having only C. In fact, POSIX specifies the behavior of rename(const char *old, const char *new); syscall that "if new names an existing directory, it shall be required to be an *empty* directory." A similar piece of information is given in the Linux programmer's manual: "old can specify a directory. In this case, new must either not exist, or it must specify an *empty* directory." This means that renaming of an empty directory C to a non-empty directory A should not have succeeded. Hence, this is not only a POSIX incompliance but also a crash consistency bug, as three persisted inodes are eventually lost after a crash.

Another bug-triggering case of Yxv6 is shown in Figure 23. A new file foo is created, and some data is written to make the file size 1,947 bytes (line 3). Thenftruncate syscall is invoked on the

```
int fd = open("./fifo", O_RDWR, 0); // fifo: FIFO special file in the base image
chmod("./fifo", 0400);
fsync(fd); // permission mode (0400) of fifo should be persisted
```

Fig. 24. ext4- fsync fails to persist metadata of a FIFO special file foo. This is bug 1 in Table 6 and Figure 19.

```
unsigned char buf[111] = {0, };
int fd_root = open(".", O_DIRECTORY, 0);
link("foo/bar/baz", "./x"); // foo bar, and baz already exist in the base image
fsync(fd_root);
int fd = open("./x", O_RDWR, 0);
mkdir("A", 0777);
setxattr("A", "system.advise", buf, 111, XATTR_CREATE);
link("foo/bar/xattr", "A/y"); // xattr is another already existing empty file
fsync(fd);
```

Fig. 25. F2FS: Directory A becomes inaccessible after a crash. This is bug 8 in Table 6 and Figure 19.

file descriptor of foo, enlarging the file size to 2,791 bytes. Although we omitted in the PoC, the return value of ftruncate is 0, which is a success code. As the PoC ends with a call to fsync, Yxv6 should guarantee that the new size (2,791 bytes) is persisted on a disk. However, the size of foo found after a crash is 1,947, meaning that it is not crash consistent.

The developers of Yxv6 have confirmed both cases yet claimed that neither the specification nor the verification was broken, but their unverified glue code for the FUSE wrapper had bugs that caused these behaviors. Still, such instances effectively show that even for the formally verified systems, Hydra is a practical approach to explore and reveal bugs in the end product of the file system that end users install and use.

6.2 Crash Consistency Bugs in In-Kernel File Systems

In this section, we analyze the remaining crash consistency bugs found in ext4, Btrfs, and F2FS.

Ext4. Hydra revealed one bug from one of the most widely used Linux file systems: ext4. The bug, shown in Figure 24, is simple; when the metadata of a special FIFO file is changed (permission mode in this case), it is not flushed to disk even with an explicit call to fsync. Currently, this is true for all special files in Linux, such as block device files, or sockets, because the VFS layer does not have fsync function defined for these special files, and thus calling fsync on these files becomes a no-op. Nevertheless, the Linux man page states that "As well as flushing the file data, fsync() also flushes the metadata information associated with the file (see inode(7))," where *file* could refer to any special file. Regarding this, the ext4 developer noted the following:

> "We should either fix the man page to document existing practice, or change the kernel" [57].

Btrfs. We have already presented one case in Figure 1, which corresponds to bug 2 of Figure 19. By tracking down the patch[7] for this bug, we found that the bug was inadvertently introduced from a fix of another old crash consistency bug. As we pointed out in Section 5.7, this case shows how Hydra complements the blind spot of regression testing. Meanwhile, the other consistency issues in Btrfs are not patched yet.

F2FS. Hydra detected the bug in Figure 25 in kernel v4.18, and it was already fixed before v4.19 was released. This case demonstrates how complicated the sequence of syscalls can be to trigger a crash consistency bug. The sequential execution of syscalls creating a hard link x of an existing file

[7] https://patchwork.kernel.org/patch/10837829/.

```
1 int fd = open("./A", O_DIRECTORY, 0); // dir A exists in the base image
2 chmod("./A", 0777);
3 fsync(fd);
```

Fig. 26. F2FS: fsync fails to persist permission of directory A. This is bug 7 in Table 6 and Figure 19.

foo/bar/baz, making a new directory A and setting an extended attribute, and creating another hard link A/y of an existing file foo/bar/xattr under this new directory, collaboratively triggers a crash consistency bug, and directory A becomes inaccessible after a crash. At the same time, the link count of file foo/bar/xattr remains as two. This is another crash inconsistency because its hard link A/y no longer exists in the image as a side effect of directory A being inaccessible. As shown in the example, it is challenging to reason about how each syscall affects the file system to eventually trigger the bug and what the result of bug will be. This shows the worth of Hydra in terms of generating a bug-triggering input that humans can hardly formulate.

In the meantime, some cases are remarkably simple, as shown in Figure 26. A change in mode is not persisted through fsync, resulting in an inconsistency in face of a crash. From the patch, we found that the bug was caused because f2fs_setattr function missed adding an inode into global dirty list once the mode is changed, and so fsync did not flush the metadata of this inode. This test case is now added to the regression test suite of F2FS.

7 DISCUSSION

Even though Hydra is capable of detecting a large portion of file system bugs (semantic and memory bugs), the current input generation and execution model of Hydra is best suited for the single-threaded, single-node environment. To support the detection of various concurrency bugs, besides plugging in a proper checker (e.g., KTSan [18]), improvements on the input mutator are required to generate inputs with interleaved operations to be run on multiple threads. In addition, a proper coverage metric to capture the code coverage and the context regarding the interleavings of multiple threads will better drive the fuzzer to buggy conditions.

Another possible future direction is extending Hydra beyond a single node to support network-based file systems (netfs) that typically run with multi-node, client-server configurations (e.g., NFS, Samba, and Ceph). Interesting research questions arise from this direction, such as how to capture the code coverage across nodes with different roles, or how to generate and mutate network events and topology changes.

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: model checking and formal verification.

First, model checking can be done in a static or dynamic fashion. Static approaches, such as JUXTA [37] and FERRITE [5], attempt to compare either well-specified [5] or inferred [37] models with the design or implementation of file systems. Since static checkers lack concrete execution states, they fundamentally suffer from high false positives, such as having stochastic errors in inferred models [37].

Dynamic approaches, such as FiSC [61] and eXplode [59] for general storage system bugs, B3 [39] for crash consistency, and SibylFS [49] 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 of the possible non-deterministic executions.

Second, formal verification [9, 10, 32, 54] is a promising new direction that can, in theory, eliminate all of 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 [25], use-before-initialization vulnerabilities [34], or even bugs in deep learning systems [43]. As described in Section 3.4.2, Hydra can readily be extended to find these bug types by providing corresponding checkers for existing file systems.

Fuzzing kernels. Syzkaller [20], kAFL [52], TriforceAFL [41] 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 article 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, whereas Hydra takes care of file system state exploration and test minimization. In our prototype, we equipped Hydra with both homegrown and external bug checkers and discovered 11 crash inconsistencies, four POSIX violations, 23 logic bugs, and 123 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.

ACKNOWLEDGMENTS

We thank the reviewers, Tej Chajed, Theodore Ts'o, and Junfeng Yang, for their insightful comments.

REFERENCES

[1] Josef Bacik. 2017. Btrfs: Add a Extent Ref Verify Tool. Retrieved April 10, 2020 from https://patchwork.kernel.org/patch/9978579/.

[2] Wendy Bartlett and Lisa Spainhower. 2004. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 87–96.

[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'17)*.

[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'16)*.

[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'16)*. 83–98.

[6] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. 2007. Ext4: The next generation of Ext2/3 filesystem. In *Proceedings of the USENIX Linux Storage and Filesystem Workshop*.
[7] Tej Chajed. 2018. FSCQ Developer's Comment on Logged Writes (Git Commit). Retrieved April 10, 2020 from https://github.com/mit-pdos/fscq/commit/97b50eceedf15a2c82ce1a5cf83c231eb3184760.
[8] Tej Chajed. 2019. FSCQ Developer's Comment on Fdatasync (GitHub Issue). Retrieved April 10, 2020 from https://github.com/mit-pdos/fscq/issues/14#issuecomment-485482506.
[9] 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'17)*.
[10] 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'15)*.
[11] 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'04)*.
[12] 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'14)*.
[13] 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'12)*.
[14] 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)*.
[15] Google. 2016. KernelAddressSanitizer, a Fast Memory Error Detector for the Linux Kernel. Retrieved April 10, 2020 from https://github.com/google/kasan.
[16] Google. 2018. KernelMemorySanitizer, a Detector of Uses of Uninitialized Memory in the Linux Kernel. Retrieved April 10, 2020 from https://github.com/google/kmsan.
[17] Google. 2018. Syzbot. Retrieved April 10, 2020 from https://syzkaller.appspot.com.
[18] Google. 2015. KernelThreadSanitizer, a Fast Data Race Detector for the Linux Kernel. Retrieved April 10, 2020 from https://github.com/google/ktsan.
[19] Google. 2019. Honggfuzz. Retrieved April 10, 2020 from http://honggfuzz.com/.
[20] Google. 2019. Syzkaller Is an Unsupervised, Coverage-Guided Kernel Fuzzer. Retrieved April 10, 2020 from https://github.com/google/syzkaller.
[21] Bogdan Gribincea. 2009. Ext4 Data Loss. Retrieved April 10, 2020 from https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.
[22] 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'07)*.
[23] 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'17)*.
[24] 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'18)*.
[25] 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)*.
[26] Dave Jones. 2018. Linux System Call Fuzzer. Retrieved April 10, 2020 from https://github.com/kernelslacker/trinity.
[27] Jan Kara. 2014. ext4: Forbid Journal_async_commit in Data=ordered Mode. Retrieved April 10, 2020 from https://patchwork.ozlabs.org/patch/414750/.
[28] Kernel.org Bugzilla. 2018. Btrfs Bug Entries. Retrieved April 10, 2020 from https://bugzilla.kernel.org/buglist.cgi?component=btrfs.
[29] Kernel.org Bugzilla. 2018. Ext4 Bug Entries. Retrieved April 10, 2020 from https://bugzilla.kernel.org/buglist.cgi?component=ext4.
[30] Michael Kerrisk. 2019. Fsync, Fdatasync—Synchronize a File's In-Core State with Storage Device. Retrieved April 10, 2020 from http://man7.org/linux/man-pages/man2/fdatasync.2.html.
[31] 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 (SOSP'19)*.

[32] Eric Koskinen and Junfeng Yang. 2016. Reducing crash recoverability to reachability. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL'16)*.

[33] LLVM Dev Team. 2019. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. Retrieved April 10, 2020 from https://llvm.org/docs/LibFuzzer.html.

[34] 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'17)*.

[35] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A study of Linux file system evolution. *ACM Transactions on Storage* 10, 1 (Jan. 2014), Article 3, 32 pages. DOI: https://doi.org/10.1145/2560012

[36] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. BugBench: Benchmarks for evaluating bug detection tools. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, Vol. 5.

[37] 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'15)*.

[38] MITRE Corporation. 2009. CVE-2009-1235. Retrieved April 10, 2020 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1235.

[39] 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'18)*.

[40] Ingo Molnar and Arjan van de Ven. 2019. Runtime Locking Correctness Validator. Retrieved April 10, 2020 from https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[41] NCC Group. 2017. AFL/QEMU Fuzzing with Full-System Emulation. Retrieved April 10, 2020 from https://github.com/nccgroup/TriforceAFL.

[42] 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*.

[43] 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'17)*.

[44] 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)*.

[45] 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'05)*.

[46] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux kernel library. In *Proceedings of the 9th Roedunet International Conference (RoEduNet'10)*. IEEE, Los Alamitos, CA.

[47] 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'17)*.

[48] Red Hat Inc. 2018. Utilities for Managing the XFS Filesystem. Retrieved April 10, 2020 from https://git.kernel.org/pub/scm/fs/xfs/xfsprogs-dev.git.

[49] 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'15)*.

[50] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. ACM Transactions on Storage 9, 3 (2013), Article 9.

[51] Andrey Ryabinin. 2014. UBSan: Run-Time Undefined Behavior Sanity Checker. Retrieved April 10, 2020 from https://lwn.net/Articles/617364/.

[52] 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*.

[53] GitHub. 2018. Linux Test Project. Retrieved April 10, 2020 from https://github.com/linux-test-project/ltp.

[54] 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'16)*.

[55] Silicon Graphics Inc. (SGI). 2018. (X)fstests Is a Filesystem Testing Suite. Retrieved April 10, 2020 from https://github.com/kdave/xfstests.

[56] Theodore Ts'o. 2018. Ext2/3/4 File System Utilities. Retrieved April 10, 2020 from https://github.com/tytso/e2fsprogs.

[57] Theodore Ts'o. 2019. Ext4 Developer's Comment on Fsync and Special File. Retrieved April 10, 2020 from https://bugzilla.kernel.org/show_bug.cgi?id=202485#c3.

[58] 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)*.

[59] 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'06)*.

[60] 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)*.

[61] 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'04)*.

[62] Chao Yu. 2018. F2fs: Disable F2fs_check_rb_tree_consistence. Retrieved April 10, 2020 from https://lore.kernel.org/patchwork/patch/953794/.

[63] Michal Zalewski. 2014. Bash bug: The Other Two RCEs, or How We Chipped Away at the Original Fix (CVE-2014-6277 and'78). Retrieved April 10, 2020 from https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html.

[64] Michal Zalewski. 2019. American Fuzzy Lop (2.52b). Retrieved April 10, 2020 from https://lcamtuf.coredump.cx/afl.

[65] Andreas Zeller, Holger Cleve, and Stephan Neuhaus. 2019. Delta Debugging: From Automated Testing to Automated Debugging. Retrieved April 10, 2020 from https://www.st.cs.uni-saarland.de/dd/.

Received January 2020; revised March 2020; accepted March 2020