LFuzz: Exploiting Locality-Enabled Techniques for File-System Fuzzing

Wenqing Liu(✉) and An-I Andy Wang

Florida State University, Tallahassee, FL 32306, USA
{liu,awang}@cs.fsu.edu

Abstract. File systems (FSes) store crucial data. However, FS bugs can lead to data loss and security vulnerabilities. FS fuzzing is an effective technique for identifying FS bugs that may be difficult to detect through traditional regression suites and human testing. FS fuzzing involves two parts: (1) File image fuzzing often involves altering bits of an FS at random storage locations; (2) File operation fuzzing typically issues random sequences of file operations to an FS image.

Since leading FS fuzzers tend to access a *small* set of files to encourage the exploration of deep code branches, the accessed FS image locations tend to be clustered and localized. Thus, altering bits at random FS locations is ineffective in triggering bugs, as these locations are often not referenced by file operations. Furthermore, the minimum FS image is insufficiently small for frequent image saves and restores due to performance and storage overhead.

In this paper, we introduce LFuzz, which exploits the locality shown in typical FS fuzzing workloads. LFuzz tracks recently accessed image locations and nearby locations to predict which locations will soon be referenced. The scheme is adaptive to migrating file access patterns. Moreover, since modified image locations are localized, LFuzz can compactly and incrementally accumulate FS image changes so that FS states can be fuzzed from intermediary images instead of top-level seed images. LFuzz further explores the use of partially updated images to simulate corrupted FSes with mixed versions of metadata.

We applied LFuzz to ext4, BTRFS, and F2FS and found 21 new bugs. Compared to JANUS, LFuzz reduced the fuzzing area by up to 8x with unique edges deviated by up to 15%.

Keywords: File-system · Locality · Fuzzing

1 Introduction

File systems (FSes) are important for holding consistent and persistent data and metadata, or FS ***states*** that survive reboots and crashes. FS bugs can have negative consequences, ranging from deadlocking and crashing an operating system to losing data and exposing security vulnerabilities. An adversary can lure a user to mount a crafted FS storage image via a swapped or misplaced USB device or a malware-infected USB charging port [3, 7]. An adversary can also issue a sequence of file operations that lead to vulnerabilities or the escalation of privileges [13].

G. Tsudik et al. (Eds.): ESORICS 2023, LNCS 14345, pp. 507–525, 2024.
https://doi.org/10.1007/978-3-031-51476-0_25

Traditional FS bug identification methods rely on manual testing and regression test suites [1, 23]. However, human enumerations of testing cases may miss bugs triggered by complex constraints. One can also exhaustively test a small number of file operations (e.g., 3) [15], but it can miss bugs that involve many file operations.

An alternative is ***fuzzing***, which uses random inputs and can identify bugs that evade regression test suites. Syzkaller [26] is a popular kernel fuzzer. Running continuously, Syzkaller has identified 2,800 + bugs in 2.5 years to upstream Linux kernels. Other kernel fuzzers (e.g., kAFL [22] and Syzkaller derivatives [28]) can also detect > 8 new bugs within days of fuzzing, indicating that fuzzing is promising for exploring difficult corner-case bugs.

FS fuzzing has unique properties since an FS accepts two streams of inputs—file operations and the stored bits that hold the content and states of an FS or an ***FS image***. ***FS image fuzzing*** often involves altering bits at random locations of an FS image. ***FS operation fuzzing*** typically involves applying random sequences of file operations to an FS image. There are two challenges. First, even though the minimum sizes of FSes are small (8MB to 128MB) compared to modern storage, leading FS fuzzers tend to avoid saving and restoring FS images (states) across fuzzing iterations due to prohibitive performance and storage overheads. The avoidance methods generally involve regenerating FS images or reducing the reproducibility of bugs.

Another challenge is assuring that file operations access the fuzzed FS image areas. For example, fuzzing file X's metadata will not affect the FS code execution branch coverage if the file operations only reference file Y's metadata. One solution is to trace accessed FS image regions for a sequence of file operations, fuzz these regions, and replay the file operations. However, accessing a fuzzed bit during one operation can alter the accessed FS stored regions for subsequent file operations. For instance, if the allocation bit for the first metadata slot is fuzzed and marked as allocated, then the next file creation (e.g., file Z) will allocate the second slot. Fuzzing the first slot based on the trace is not effective, as file Z's operations will reference the second slot.

We designed, implemented, and evaluated LFuzz, an FS fuzzing framework, to address these challenges. We observed that FS fuzzers typically access a small set of files (e.g., < 100 within 240 CPU fuzzing hours) to encourage deeper state explorations, even for an FS with many files. This means that the accessed FS image locations can be clustered and localized. Thus, fuzzing recently accessed and neighboring image locations can increase the probability that the next file operations will access those locations. The locality of the FS image updates also leads to smaller and clustered modified image ranges, reducing the overhead for saving and restoring FS images in incremental deltas. Additionally, we discovered that incompletely restoring deltas emulates an FS with mixed metadata versions, which is another effective fuzzing method.

We applied LFuzz to ext4 [12], BTRFS [21], and F2FS [9] for 240 CPU hours. Compared to JANUS [30], unique edges explored by LFuzz deviated from those of JANUS by up to 15%, and FS fuzzing area was reduced by up to 8x. Furthermore, LFuzz discovered 21 new bugs.

2 Background

FS basics: Each file is associated with an ***i-node***, which is a per-file data structure. The allocation of i-nodes and data ***blocks*** (1KB-8KB) may involve allocation ***bitmaps*****,** in which each bit indicates whether an i-node or a data block is allocated. A ***directory*** maps file names to i-node numbers. A single file operation can update multiple data structures. For example, moving a file from one directory to another directory involves changes to both directories). To make such operations appear indivisible, an FS may provide a ***journal*** to record multiple operations in a transaction. Each FS also has a ***superblock***; that provides global information about the FS type, the total number of free blocks, etc. File content is referred to as ***data***, while the remaining data structures (e.g., i-nodes) are referred to as ***metadata***.

Some FSes use ***copy-on-write*** (***COW***) mechanisms. Instead of making updates in place, COW FSes write updates to unwritten locations with a version stamp. An application program issues file operations to an FS through ***system calls*** (or ***syscalls***, for short). A block-based FS typically accesses storage devices through a ***block layer***, which translates file-level requests into block-level requests (e.g., block writes). To optimize performance, a referenced FS block can be cached in a memory ***page*** (1KB-8KB, typically the same as the block size) to accelerate future access to the same block.

Leading FS fuzzers: *Syzkaller* [26] creates an FS image by picking a parameter set and prefilling the FS with files and directories. For each fuzzing iteration, a random sequence of fuzzed file operations is applied. The random file operations follow FS semantics (e.g., a file write is issued only to an opened file [27]). The test directories are deleted after each iteration. The number of files being fuzzed can be limited. Within 240 CPU fuzzing hours, Syzkaller references up to four files on average within each iteration, with up to 13 file operations in a sequence. The average number of operations on the files is two.

Syzkaller generally does not save FS images (except syz-mount-image fuzzing). File operation sequences are tested one after another without resetting the kernel until a time limit or until the container VM needs to reboot. Therefore, when a bug is detected (e.g., system crashes, kernel panics, BUG() and KASAN [5] error messages, time outs), it is difficult to discern whether the bug was caused by the last file operation sequence or the cumulative FS state changes up to that point. From our experience, when the last file operation sequence is applied to the original image, only 50% of the bugs can be reproduced. Xu et al. [30] discovered that all crash-related bugs for Syzkaller are not reproducible.

AFL [32] has been used to fuzz FS images [17], which can be used to run regression tests. AFL fuzzes only nonzero metadata blocks because data blocks generally do not affect FS integrity. Therefore, AFL may skip valid metadata blocks that are zero-initialized. Moreover, for COW FSes, obsolete nonzero metadata blocks can dilute fuzzing targets because fuzzed obsolete blocks are unlikely to be referenced and contribute to identifying new execution branches.

JANUS [30], which is based on AFL, fuzzes file operations and FS images. To reduce the FS image area to be fuzzed, JANUS extracts the image regions that are initially allocated for metadata with prepopulated files and fuzzes only that fixed region. Whenever JANUS idenfities new code execution branch coverage after applying file

operations, it saves a new seed image by recording the file operation sequence and the old seed image *before* the operations are applied. The new seed image can be regenerated by applying the saved file operation sequence to the old image.

JANUS also handles blocks with checksums, such as superblocks. Fuzzing a superblock likely leads to mount failures, precluding the exploration of deeper code branches beyond checksum verification. Thus, JANUS makes checksums consistent with fuzzed content, simulating corruptions before the checksums are computed.

3 Image Access Locality of FS Fuzzers

Accessed FS image locations differ based on the seed image and migrate over time. The challenge in deciding which FS image locations to fuzz lies in predicting how accessing the fuzzed bits will alter subsequent access locations. To address this issue, we examined the size of the accessed areas for a fuzzing iteration, their temporal relationship across fuzzing iterations, and their interactions with structured FS layouts.

Size of accessed FS image locations: By intercepting bio_endio() at the block layer, we traced the FS locations (in 64B subblocks or ***buckets***) accessed by 200 random file operations issued by JANUS and aggregated the accessed size. The results for ext4, BTRFS, and F2FS are shown in Table 1. The total accessed size by these operations was $< 0.02\%$ of the smallest FS image. Although JANUS narrowed the fuzzing to the initial metadata regions, the accessed image size was still $< 13\%$ of the reduced region, indicating that ***the chance of random file operations accessing a randomly fuzzed region was small*** ($<13\%$ even when fuzzing is limited to the initial metadata). This finding also implies that ***the overhead of tracking, saving, and restoring just the accessed FS image locations might be affordable***.

Table 1. Size of image locations accessed by 200 random file operations issued by JANUS.

	ext4	BTRFS	F2FS
Smallest FS image	8MB	128MB	64MB
Initial metadata size fuzzed under JANUS	111KB	41KB	90KB
Accessed image size by 200 random file operations	1.3KB	3.3KB	12KB

Temporal correlations of accessed image locations: During the image fuzzing phase of JANUS, the same file-operation sequence was repeatedly applied in iterations to the same FS image fuzzed in different ways. One hypothesis was that the image locations accessed in one fuzzing iteration would likely correlate with the next fuzzing iteration.

To test this hypothesis, we modified JANUS to trace accessed image locations from one iteration to the next iteration during the image fuzzing phase. After 6K iterations, we discovered that, for ext4, 78% of the accessed image locations for one iteration overlapped with the accessed image locations of the next iteration. Similarly, the overlapping rates for BTRFS and F2FS were 75% and 80%, respectively. Thus, ***fuzzing the current***

accessed image locations led to a high probability that they would be accessed by the next fuzzing iteration.

Spatial correlations of referenced image locations: Since stored FS metadata blocks are not randomly allocated (e.g., sequentially and hierarchically), we examined whether there is a block distance relationship between the updated 4KB blocks from one iteration to the next iteration. Assume that iteration one updated blocks 1 and 2 and iteration two updated blocks 3 and 4. We computed all pair-wise distances from the newly referenced blocks from the second iteration to the blocks from the first iteration: (3 - 1), (3 - 2), (4 - 1), and (4 - 2). Thus, we had 2, 1, 3, and 2. The newly accessed block had a 50% chance of being two blocks from any blocks in the first iteration and a 25% chance of being one or three blocks away from any blocks in the first iteration. We bound the distance to 50 blocks. Since an update might involve different metadata structures located in different areas (e.g., journal and i-node blocks), the distance between these areas minimally reflects how metadata blocks of the same type are allocated.

We ran modified JANUS to fuzz 64B buckets accessed from the previous iteration for two hours and measured the distance between updates in blocks (Figs. 1, 2 and 3). For ext4, the most popular update neighbor distance was 1, indicating that the blocks were sequentially allocated. In addition, when fuzzing the next iteration, the next referenced blocks were likely to be within three blocks of a block referenced within the current iteration. For BTRFS, the range was more scattered due to the use of b-trees. For F2FS, the most popular update neighbor distances are 1, -1, and 5. These results indicate that ***fuzzing the neighbors of the accessed blocks for this fuzzing iteration can increase the chance that these blocks will be accessed in the next iteration***.

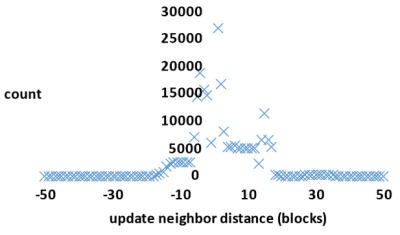

Fig. 1. Frequency of distances between updated blocks across iterations for ext4.

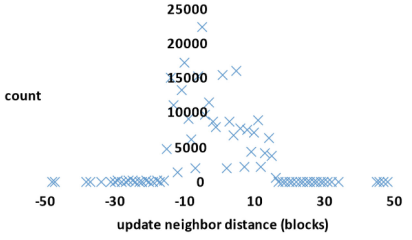

Fig. 2. Frequency of distances between updated blocks across iterations for BTRFS.

4 LFuzz Design

Although locality is used to optimize FSes, it is counterintuitive to apply locality to fuzzing because fuzzing thrives on the use of random inputs to increase code execution branch coverage. Based on the findings in Sect. 3, we designed and implemented LFuzz, an FS fuzzing framework that exploits spatial and temporal localities. Regarding temporal locality, LFuzz fuzzes currently accessed image locations, since the next fuzzing iteration will likely have overlapping accessed locations. Regarding spatial locality, LFuzz fuzzes neighboring blocks of currently accessed image locations, since the next fuzzing iteration will likely access surrounding locations.

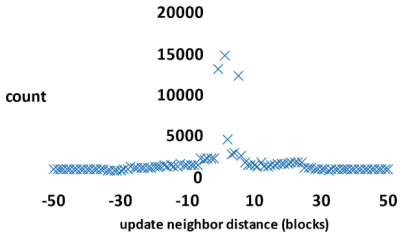

Fig. 3. Frequency of distances between updated blocks across iterations for F2FS.

LFuzz maps 4KB memory pages to FS storage blocks when they are cached in memory. LFuzz also tracks 64B memory accesses to identify the corresponding accessed 64B FS subblock regions. Accessed FS image locations are stored in a least-recently-used (LRU) list with a bounded length to adapt to locality changes. For each iteration, LFuzz fuzzes these accessed FS locations and some neighboring locations. Thus, when an FS image is saved and restored, so is the LRU list. We utilized 64B list elements (< 4KB blocks) to reduce the storage and saving/restoration overhead.

To reduce the cost of saving FS images, we also introduced the notion of deltas, which were obtained by subtracting modified image I' from image I before applying fuzzed file operations. The deltas are captured in 256B subblocks (< 4KB block) to reduce the storage and saving/restoration overhead.

In addition, we found that partially restored deltas could lead to many FS bugs. A partially restored delta creates two FS image areas, each of which is largely self-consistent (e.g., each area has correct checksums for i-nodes); however, these areas are globally inconsistent. Therefore, we incorporated this technique into our delta fuzzing.

We adopted the framework from JANUS, which used the Linux Kernel Library (LKL) [20] that enables the Linux kernel to be compiled as a user-level library. LKL can be used by fuzzers to fuzz kernel FS code in the user space. JANUS stored FS images in memory as opposed to on disks and solid-state devices (SSDs), so we could replace slow system reboots with resets to the memory-resident FS image. All fuzzing states were also memory-resident.

4.1 Fine-Grained Tracking of Accessed FS Storage Image Locations

To track fine-grained (64B) access of FS storage image locations, it is insufficient to intercept FS traffic to and from storage devices, where fined-grained accesses are aggregated into 4KB block access units. We also need to intercept fine-grained memory accesses to cached FS storage blocks to identify fine-grained storage accesses. One implication is that we need to map accessed addresses of memory-cached FS blocks to 64B FS storage locations via an M2F table (Table 2).

LFuzz instruments the block layer. As FS blocks are read into the memory, LFuzz creates the mapping of memory pages to FS storage blocks. To create a fine-grained mapping, we also need to determine which 64B area of a 4KB memory page is accessed. We injected code during the compilation stage (details in Sect. 5), which reports accessed FS memory addresses to capture this fine-grained information.

For example, if the memory address 0x7FF0041a3088 was accessed, it would be within 0x7FF0041a3000 + memory page size (0x1000 bytes); thus, based on the M2F table, FS storage image block 9217 was accessed. Since the accessed memory address belongs to the 2nd 64B area or bucket of a 4KB memory page, the 2nd bucket of the corresponding 4KB FS storage block is accessed. Once the block number and bucket number are identified, it is stored in the LRU bucket list (Table 3).

Table 2. Memory Address to FS Storage Image Block (M2F) Mapping Table.

Memory address	Image block number
0x7FF004195000	5121
0x7FF0041a3000	**9217**
…	…

Table 3. LRU List.

Image block number	Bucket number	Timestamp
9221	17	1
9217	**0**	**16**
…	…	…

4.2 Exploiting Locality for Fuzzing

Temporal locality: LFuzz exploits temporal locality by using an LRU list to track recently accessed image locations as potential targets for fuzzing. This is particularly helpful when the locality changes over time. Metadata blocks may be dynamically allocated beyond the initial metadata regions, and the LRU list can adapt to the workload and include dynamically allocated blocks for fuzzing.

In addition, the LRU list is bounded so that less frequently accessed locations will be dropped as potential fuzzing targets. COW FSes may leave obsolete blocks behind and dilute the fuzzing targets. Dropping obsolete blocks from the fuzzing targets increases the probability that fuzzed locations will be referenced in the next iteration.

Spatial locality: Accessed image locations change when metadata areas are fuzzed. For instance, when a file creation request is issued, an FS needs to assign an unused i-node to the created file. If the i-node bitmap is fuzzed to mark some i-nodes as used, then the newly allocated i-node will skip those entries. To increase the probability that fuzzed image locations will be referenced in the next fuzz iteration, LFuzz chooses neighboring and referenced locations as fuzzing targets.

Intra-block locality: Since the image fuzzing phase typically involves a few files with a limited number of operations in a testing sequence, i-nodes, directory entries,

and other metadata tend to be allocated in succession. Thus, an allocation is likely to be fulfilled with nearby free space. With a 64B bucket size, we discovered that > 75% of image bucket locations accessed in this fuzz iteration were accessed in the next fuzz iteration. Thus, fuzzing the currently accessed image locations and the surrounding locations increases the probability that these locations will be referenced in the next fuzz iteration.

Inter-block locality: Since metadata blocks can be allocated in succession, neighboring blocks will likely be accessed in the next fuzzing iteration. For example, a COW FS may write a metadata block to another (potentially neighboring) metadata block. Thus, if LFuzz accessed a bucket with a particular offset within the current block, we will add buckets with the same offset from neighboring blocks as potential targets for fuzzing. For example, if in the current fuzzing iteration, ext4 accesses byte 8 of block 51, in the next iteration, LFuzz will fuzz bucket 0 containing bytes 0–63 in block 51, and its neighboring block $51 + 1 = 52$, where one is the most frequently occurred block neighboring distance.

4.3 Image Deltas

Given the locality in fuzzing workloads, LFuzz uses image deltas to reduce the FS image storage and saving/restoration overhead. Image delta D is defined as modified FS image I' subtracted from original image I_0 (before modifications). This subtraction can be expensive if only a few places are modified due to locality. Thus, LFuzz applies a COW mechanism to I_0 so that only modified image regions are copied and tracked.

Unlike leading FS fuzzers, which save images when new coverage or bugs are discovered, LFuzz saves lightweight delta images whenever an FS image is modified. Therefore, instead of replaying file operations from the top-level image, a newly fuzzed file operation only needs to be applied to the saved delta image, which has accumulated the FS state changes for all proceeding file operations.

Note that LFuzz only accumulates the differences in FS images; dynamic states, which track whether a file is open and the file pointer position, are excluded. To illustrate this point, the behavior of opening a file and writing a block to that file is different from that of opening a file, saving and restoring a delta image, and writing a block to that file. In the latter case, writing a block is expected to fail (not a bug) because the file has not been opened after an image restoration. Thus, the behavior of the restored LFuzz image deviates from the behavior of restoring an image regenerated from the seed image by applying all operations in a sequence, which also restores dynamic states.

While not saving and restoring dynamic states, deltas can accumulate modified states at a faster rate. For example, if an FS image is repeatedly fuzzed and tested with the same file operation sequence that creates random files, created files and various updates accumulate and stay from one fuzzing iteration to the next instead of being reset via regeneration from the seed image. Unlike Syzkaller, LFuzz reproduces bugs by applying the latest file operation sequence to the latest delta instead of the root image. LFuzz's bug reproducibility rate is ~ 85%.

4.4 Partially Updated Images

While developing the image delta technique, we found that partially restored deltas led to FS bugs and crashes. Further investigation revealed that partially restored deltas emulated the scenario in which two versions of FS states coexisted. Thus, segments of the FS states were self-consistent, while, globally, the FS states were inconsistent. Since we identified quite a few bugs in this way, we incorporated this fuzzing technique into delta fuzzing. Based on empirical experience, we set the probability of triggering a partial update as the current length of syscall sequence L divided by (L + 5). Since the average syscall length is ~ 30–60, the probability of triggering a partial write is ~ 90%. Initially, with shorter syscall sequences, the fuzzing coverage can grow with just delta fuzzing. As the lengthening of syscall sequences with delta fuzzing finds fewer new execution branches, the partial-write technique is more frequently triggered.

```
1  for corpus C = {image I, file ops F, LRU L from the first iteration}
2    L' = fuzzed L
3    for (iteration j < bound B) {
4      I' = apply L' to I
5      Apply F to I'
6      if (new coverage found}{
7        save(I', F, L')
8        if  (B < max_bound) {
9          B *= 2;
10       }
11     }
12     L' = fuzzed L'
13   }
14 }
```

Fig. 4. LFuzz LRU-based fuzzing phase.

```
1  for corpus C = {image I, file ops F, image delta D} {
2    F' = F + file ops // append random new file ops
3    D' = applying D to I
4    for (iteration j < bound B) {
5      D' = applying F' to D'
6      if (no new coverage found) {
7        move on to the next corpus (I, F', D')
8      } else {
9        save(I, F', D')
10       if  (B < max_bound) {
11          B *= 2;
12       }
13       F' = F + fuzzed file ops // append random new file ops
14     }
15   }
16 }
```

Fig. 5. LFuzz delta-image-based fuzzing phase.

4.5 LFuzz Phases

LFuzz's fuzzing order has three phases: LRU-based fuzzing, syscall fuzzing (JANUS-based), and syscall fuzzing with delta. We appied a similar fuzzing order to make LFuzz and JANUS comparable. Syscall fuzzing was needed since an FS needs file operations to change FS image states. Another similarity is that the next fuzzing phase is only triggered when the current phase cannot find any new coverage.

Figure 4 presents the LRU-based fuzzing phase, in which the same syscall sequence is applied to the same image fuzzed differently. At line 1, a new corpus is loaded with initial image I, sequence of file operations F, and LRU regions L from the first fuzzing iteration. A subset of buckets L' in LRU L are fuzzed (line 2), and L' is applied to the original image I (line 4) to form fuzzed image I'. The sequence of file operations F is applied to fuzzed image I' (line 5). If new coverage is found, modified image I', sequence of file operations F, and modified LRU regions L' are saved (line 7). The number of fuzzing iterations (bound B) is decided by AFL's initial execution speed, which is the fuzzing framework JANUS is based on. In our experiment, B is initialized to 128. B can be increased is based on when new coverage is found (lines 8–9). At the end of the iteration, LRU regions L' are fuzzed for the next iteration (line 12).

Figure 5 presents the delta-based syscall fuzzing phase, in which an incrementally increased file operation sequence is applied to delta images updated after each iteration. At line 1, a new corpus is loaded with initial image I, sequence of file operations F, and saved delta image D. Sequence of file operations F is appended with randomly selected file operations to form F' (line 2). Fuzzed delta image D' is formed by applying delta region D to initial image I (line 3). The newly formed delta image D' is updated by applying the new sequence of file operations F' to itself to accumulate states for each iteration (line 5). If new coverage is not found, LFuzz moves on to the next temporary corpus (with image updated to the I + D') (line 7). If new coverage is found, initial image I, updated sequence of file operations F', and updated delta image D' (line 9) are saved, and the number of iterations is increased (lines 10–11). Finally, randomly selected file operations are appended to F' for the next iteration (line 13).

5 Implementation

We chose to build LFuzz on top of JANUS instead of Syzkaller because of the reproducibility of crash-based bugs and because JANUS' syscall generation follows FS semantics. Since JANUS is based on the LKL, which is infrequently updated, we had to port the Linux kernel from Linux 5.3 to 5.15. Furthermore, we based our tests on lkl-5.15 (LKL with Linux 5.15).

Predicting accessed locations for fuzzing: We added a stub to the `bio_endio()` function in `block/bio.c`, which was called after the `bio` request was finished, and the memory addresses were assigned to the `bio` structure. Then, we used LLVM to instrument the stub function to obtain the storage block information that had been loaded to the memory, including the storage offset, total loaded size, and memory addresses of cached FS content. In addition, `load` and `memcpy` LLVM intermediary representations were instrumented to obtain the fine-grained storage accessed locations by comparing the `load`ed or `memcpy`ed source memory addresses with the memory addresses of cached

FS content. Knowing the memory address of cached FS content, the instrumentation can also filter out memory accesses that do not affect the FS image. With the fine-grained accessed locations, LFuzz maintains an LRU bucket list to exploit localities and generate candidate FS bucket locations for fuzzing. After fuzzing the FS image, LFuzz also fixed various checksums as JANUS.

Capturing deltas: To incrementally track FS image updates, instead of instrumenting the bio_endio() function, LFuzz leveraged JANUS' page-fault handler in `userfaultfd` to capture accesses to FS images that triggered page faults. The reason is that we only need to compare the final state of an image after delta stage file operations are applied to the image, while bio_endio() captures the intermediate state of the image during the execution of the file operations. JANUS allocated a temporary empty buffer I' with the size of an FS image. When executing a file request sequence, JANUS (LKL) triggered a page fault whenever a memory page in I' did not hold the content of the original FS image (I_0) and copied the FS image page content to the temporary buffer page. To enhance performance, I_0 was stored in memory pages instead of storage devices.

LFuzz leveraged this mechanism to track all FS image page offsets that triggered page faults. When the execution ended, LFuzz compared the modified pages in I' with the corresponding pages in I_0 and stored the differences at 256B granularity (based on our preliminary empirical results to balance storage and comparison overhead). Therefore, if only 256B of a 4KB block was modified, we only stored 256B. For fuzzing, deltas restored to various locations of an FS image.

To perform partial updates for a file request sequence $R_1 \ldots R_n$, we gathered page fault information from R_1 to a random file request $R_{i<n}$ in the sequence. Furthermore, for page faults F_1 to F_m triggered by $R_{j<m}$, we captured page fault information from F_1 to a random page fault F_j in the page fault sequence. To perform partial updates, we replayed the file request sequence up to R_j and restored storage blocks accessed with page faults up to F_m.

Line count: We added 1,193 lines to the LLVM runtime for locality fuzzing, 239 lines to the LLVM pass, 864 lines to wrap different FSes, 264 lines to AFL, 64 lines to intercept the block layer, and 72 lines to implement delta and missing write fuzzing.

6 Evaluation

Since JANUS has demonstrated more effective coverage than Syzkaller [30], we only compared LFuzz with JANUS. We fuzzed ext4 (the default FS for Linux), BTRFS (a COW FS), and F2FS (an FS for SSDs). These are the three FSes supported by JANUS' Github, in terms of providing the code to extract the fixed initial metadata regions for JANUS fuzzing. We used two Dell Precision 7820 workstations with Intel® Xeon® Gold 5218R 40 cores with 128GB of memory. The LFuzz and JANUS tests each comprised 10 processes. LFuzz used the default test seed from JANUS. LFuzz and JANUS tested the following syscalls: read(), write(), open(), seek(), getdents64(), pread64(), pwrite64(), stat(), lstat(), rename(), fsync(), fdatasync(), access(), ftruncate(), truncate(), utimes(), mkdir(), rmdir(), link(), unlink(), symlink(), readlink(), chmod(), setxattr(), fallocate(), listxattr(), and removexattr(). The figures were presented with a 90% confidence interval.

***Overall overhead*:** Although the overhead of instrumenting all `loads` was high to predict locality, the overhead was incurred when a new seed image was loaded or new

coverage was found. The delta state overhead mainly stemmed from comparing the deltas with the seed image, which happened during every delta fuzzing iteration. The overhead depended on the number of unique pages triggering page faults. We ran each configuration for three hours and repeated the experiments three times. The average overhead for ext4 was $< 5\%$; for BTRFS, $< 1\%$; and for F2FS, $< 15\%$. F2FS has a higher overhead because F2FS has referenced more bytes (~10 times) than the other two FSes. This outcome led to a higher overhead to copy the fuzzed content to the target image.

6.1 New Bugs

We ran LFuzz and JANUS for one week and identified 35 bugs (Appendix I), 21 of which were only identified by LFuzz. JANUS reported some crashes that LFuzz did not report, but those crashes were not reproducible. We believe the main reason that JANUS did not detect bugs that LFuzz could not find was that JANUS has been used by people to fuzz the target Fses. Therefore, we assume that most of the bugs that JANUS could identify were already fixed, and LFuzz may not be able to detect all of them. LFuzz found 13 memory bugs with security implications. The bugs were reported to Red Hat and upstream maintainers. Ten of these bugs have been patched by the maintainers, and we have requested three CVEs with assigned numbers. We will request CVEs for the other bugs in the future. Note that we are not claiming that LFuzz is better than JANUS. LFuzz and JANUS are different fuzzing strategies and explore different execution branches.

To assess the effectiveness of LFuzz's locality and delta (with built-in partial updates) fuzzing methods, we tested one method at a time. When conducting LRU locality fuzzing without deltas, we found 16 bugs (3, 7, 14–20, 23, and 28–33), including two new bugs (3 and 29). When fuzzing with deltas without locality, we found only one new bug (9). At first glance, these numbers suggested that our fuzzing methods were not effective. However, with the combined use of locality and delta fuzzing, LFuzz identified 35 bugs, including 21 new bugs (1–6, 8–13, 21–22, 24–27, 29, and 34–35).

To assess the effect of partial updates, we used locality and delta fuzzing with partial updates disabled. Eight (bugs 12–13, 21–22, 24–27, 34–35) of 21 new bugs were attributed to the presence of partial updates. Partial updates were more effective for BTRFS, a COW FS, which may have strong assumptions about FS consistency and does not interact well with partially consistent FS images.

Case study: CVE-2022–1184 (bug 4): When ext4 added a directory entry, it used `inode- > i_size` to locate or allocate the next logical directory block. The `i_size` field was affected by a corrupted FS image, and ext4 did not check whether the field was in the correct range. When `inode- > i_size` was corrupted, the computed block index could point to a block in use. Thus, the in-use block could be corrupted by this error index. In addition, if the other index pointing to the block freed the block, this index pointer would have led to a use-after-free bug. KASAN identified this problem in `do_split()`[LOR22].

Since JANUS did not accumulate files during the image fuzzing phase, we also created an almost full image for JANUS to test. However, the metadata region was too large for the AFL component to fuzz. LFuzz used locality fuzzing to reduce the fuzzing

area and track the reference migration patterns, and delta fuzzing accumulated files to fill the directory block to trigger this bug.

Case study: CVE-2021–44879 (bug 29): This bug occurred because the fuzzed image marked a data block as a special file (e.g., character, block, FIFO, and socket file). Right before the block was migrated due to the F2FS garbage collection, it invoked `a_ops-> set_dirty_page()`. However, the operation pointer was NULL for the special files, triggering a NULL pointer dereference [14].

To trigger this bug, the fuzzer needed to either modify the segment summary area (SSA) entry, pointing the migrated block's parent to a special file i-node, or fuzz the corresponding parent i-node's `imode` field as a special file. If the fuzzed i-node `imode` or SSA entries were in a state to trigger the bug, the block had to be migrated to make it happen.

JANUS fuzzed the initial metadata block regions, consuming 90KB for F2FS. During the first iteration of fuzzing, LFuzz tracked 12KB as potential fuzzing locations, which was approximately one-seventh of the size fuzzed by JANUS. Focused image fuzzing helped LFuzz detect this bug.

6.2 LFuzz Coverage

We fuzzed JANUS and LFuzz under each configuration for 240 CPU hours and compared their edge coverage, which was defined as unique edge transitions between compiled basic blocks. Specifically, we assigned each basic block a unique ID and used each unique ID pair to denote a unique edge. We defined the *unique edge deviation rate* as the total number of unique edges that were only covered by LFuzz divided by the total unique edges covered by JANUS. We ran LFuzz with and without delta (and its integrated missing write fuzzing). Based on preliminary tests, we chose the configurations that generated the best results. For the LRU fuzzing option, we tested list lengths of 512 buckets and 2K buckets, which could hold 32KB and 128KB, respectively. Each test was repeated five times and the overall edges covered by each configuration were collected. Figures 6, 7 and 8 present the edge deviation rate results for ext4, BTRFS, and F2FS. Overall, the edge coverages under various LFuzz configurations were comparable to those of JANUS. In the best cases, LFuzz explored unique edges that deviated from those of JANUS' by up to 15% for ext4, 7% for F2FS, and 3% for BTRFS with statistical significance.

We discovered that a single LFuzz configuration was unable to achieve the best coverage for all three FSes. For ext4 (Fig. 6), since the working set was smaller than 512 buckets, the edge coverage was approximately the same with a longer LRU length bound. The combination of LRU fuzzing and delta fuzzing achieved the highest coverage. For F2FS (Fig. 7), LRU fuzzing with a shorter LRU length degraded the edge coverage because the working set for F2F2 exceeded 512 buckets. Therefore, useful buckets could have been removed before the next reference. When combined with LRU, delta fuzzing increased the edge coverage. For BTRFS (Fig. 8), the variations across the configurations were within 1%. Out of curiosity, we tested delta/missing write fuzzing alone without locality fuzzing (not shown); the discovered edges were less than half of the configurations with the locality.

In general, it was difficult to attribute the cause of the coverage increase to a particular fuzz method since new coverage could be built on previously discovered and saved coverage via different fuzzing methods.

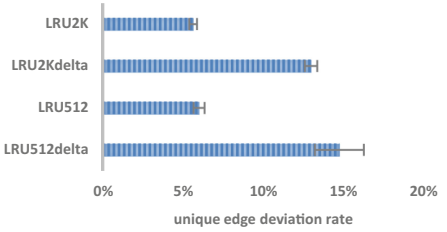

Fig. 6. Unique edges visited by LFuzz that deviated from those of JANUS' for ext4.

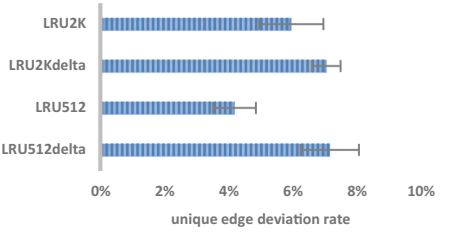

Fig. 7. Unique edges visited by LFuzz that deviated from those of JANUS' for F2FS.

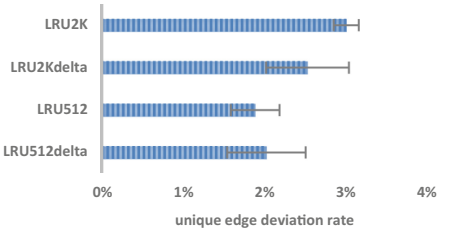

Fig. 8. Unique edges visited by LFuzz that deviated from those of JANUS' for BTRFS.

6.3 LFuzz Fuzzed Regions

Figure 9 compares the JANUS and LFuzz fuzzed region sizes during the 24th hour of the experiments, with error bars indicating the variation within one standard deviation. For ext4, LFuzz's fuzzed area was approximately one-eighth of JANUS' fuzzed area while achieving a 6% increase in the unique edge deviation rate. For BTRFS, LFuzz's fuzzed area could have been 30% smaller while achieving a 2% increase in the unique edge deviation rate. For F2FS with 2K LRU buckets, LFuzz's fuzzed area could be as large as that of JANUS, reflecting F2FS' wear leveling for its designed use on SSDs. However, with 512 buckets, LFuzz achieved a 5% increase in the unique edge deviation rate with a fuzzed area that is 34% as large as that of JANUS.

We also found that the bound on the LRU length interacted with the fuzzing results. If the bound was too large, the content held by the LRU list approached the entire working set, which contained frequently and infrequently accessed areas for fuzzing. If the bound was too small, useful content was removed before it could be soon accessed again. Based on the preliminary test results, we only systematically tested the lengths of 512 and 2K buckets to avoid the exponential explosion of the test space.

We also tried some extreme LRU length values. For example, for ext4, we tested an LRU length of 36 buckets, which could hold approximately 2KB of content. Since, for each update operation, ext4 accessed the journal last, the LRU list mostly held journal

content for fuzzing, with prior content removed due to the LRU length limit. We were able to detect a journal bug at `ext4_jbd2.h: ext4_inode_journal_mode()` after 12 h. This bug did not appear in the first 12 h when the LRU list length was longer than 64 buckets. Our future work will examine the effects of LRU list length.

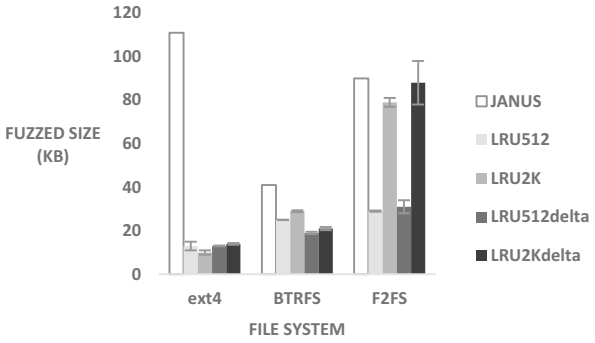

Fig. 9. Comparison of JANUS and LFuzz fuzzed region sizes under different configurations.

6.4 Fuzzing an FS Without Preknowledge of Layout

To test how easy it is to apply LFuzz to an FS without the preknowledge of the FS metadata layout (no fixing for checksums), we tested LFuzz on ntfs3. We tested LFuzz with 30 cores for three days in Linux 5.15; it found six bugs (two new) that are not fixed in the latest Linux long term (6.1.29). The first bug is kernel NULL pointer dereference in hdr_find_e, and the second bug is use-after-free in ntfs_read_hdr. We have reported these two bugs to the ntfs3 maintainer.

7 Related Work

FS fuzzers and exercisers: In addition to Syzkaller [26], AFL [17, 32], and JANUS [30] mentioned in Sect. 2, CrashMonkey [15] exhaustively tests FSes with bounded, short file operation sequences. CrashMonkey constructs FS crash states and runs FS recovery operations. It then compares the FS states to detect bugs such as incorrect file sizes and files not removed during renaming. CrashMonkey can miss bugs caused by longer file operation sequences, and LFuzz can detect memory bugs (e.g., out-of-bounds, use-after-free, protection faults, and NULL-pointer dereferencing).

Kernel fuzzers: KAFL [22] uses the Intel® processor-tracer result to guide the fuzzing to reduce overhead. HFL [6] utilizes symbolic execution to solve hard branches with complex logic. USBFuzz [19] and Periscope [24] fuzz the drivers by modifying the MMIO and DMA interfaces.

Some kernel fuzzers focus on improving the quality of syscall sequences for fuzzing. When new coverage is detected, Moonshine [18] exploits syscall read/write dependencies to filter out calls that do not contribute to the state changes of the new coverage,

thereby minimizing the length of the syscall sequence for further fuzzing. DIFUZE [2] analyzes ioctl-related code to generate valid structured input for fuzz drivers.

Instead of code coverage, some fuzzers improved the feedback strategy. SyzVegas [28] changes the seed image scheduling using multi-armed-bandit algorithms. StateFuzz [33] tracks variables that lead to state changes to prioritize test cases.

Razzer [4] uses point-to information from static analysis to generate test cases that are likely to cause race conditions. Krace [31] uses potential interleaving memory access instructions as coverage to guide the fuzzing to identify race conditions.

8 Limitations and Future Work

Since LFuzz has a large configuration space, an exhaustive exploration would involve an exponentially large number of experiments. Although we conducted many preliminary tests, we did not conduct a repeated, fine-grained exploration of different LRU lengths and their effects on various FSes. We also did not explore the effects of bucket size, delta storage granularity, the probability of triggering missing writes, or the ordering of fuzzing phases. We plan to conduct such studies in the future. Since each FS interacts with LFuzz differently, we will also explore FS-specific fuzzing.

Some code execution branches are controlled by compile time configuration, which means that fuzzing itself can never reach some code regions. We plan to explore a different fuzzing framework for fuzz compiler-enabled code branches.

Furthermore, since we fuzzed small FSes, we were unable to fuzz code branches triggered by large file sizes (e.g., 500MB). In the future, we will try to solve the hard branches with the exact referenced locations.

9 Conclusion

We have designed, implemented, and evaluated LFuzz, an FS fuzzer that exploits locality-enabled fuzzing techniques. We determined that random FS image fuzzing is insufficient because many fuzzed locations are not referenced by file operations. We analyzed the locality feature of fuzzing FS workloads on FS image modifications and proposed a locality-aware fuzzing approach for kernel FSes. Locality fuzzing is adaptive to changing reference patterns, so we do not need preknowledge of FS layouts. Our locality fuzzing scheme allowed us to perform incremental accumulation of delta states and perform partial updates. When all these methods were applied, LFuzz found 21 new bugs. In addition, we discovered that LFuzz can reduce the target fuzzing region by up to 8x compared to JANUS and visit unique edges that deviate from JANUS' edges by up to 15%.

Acknowledgements. We thank anonymous reviewers for their invaluable feedback. This work is sponsored by the National Science Foundation (DGE-2146354). Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, Florida State University, or the U.S. government.

Appendix I: Bugs Detected by LFuzz and JANUS

FS	Bug number	Bug type	Ver	Bug location	status	JANUS	LRU + delta (+ partial updates)	delta (+ partial updates)	no parial updates	LRU
ext4	1	stack-out-of-bounds	5.18	__blk_flush_plug	Ack'd	X	O	X	O	X
	2	page fault	5.18	fs/ext4/namei.c: do_split()	Ack'd	X	O	X	O	X
	3	out-of-bounds read	4.19	ext4_search_dir()	patched	X	O	X	O	O
	4	use after free	5.18	CVE-2022–1184	patched	X	O	X	O	X
	5	slab-out-of-bounds	5.18	fs/ext4/xattr.c: ext4_xattr_set_entry()	reported	X	O	X	O	X
	6	use after free	5.18	fs/ext4/namei.c:ext4_insert_dentry()	reported	X	O	X	O	X
	7	BUG()	5.18	fs/ext4/extents_status.c:202	reported	O	O	X	O	O
	8	BUG()	5.18	fs/ext4/ext4_jbd2.h: ext4_inode_journal_mode()	reported	X	O	X	O	X
	9	BUG()	5.18	fs/ext4/extent.c: ext4_ext_determine_hole()	patched	X	O	O	O	X
	10	BUG()	6.0-rc7	fs/ext4/ext4.h: ext4_rec_len_to_disk()	reported	X	O	X	O	X
	11	BUG()	5.19	fs/ext4/extents.c: ext4_ext_insert_extent()	confirmed	X	O	X	O	X
	12	NULL pointer deref	6.0-rc7	fs/ext4/ialloc.c: ext4_read_inode_bitmap()	reported	X	O	X	X	X
	13	NULL pointer deref	6.0-rc7	ext4_free_blocks()	reported	X	O	X	X	X
BTRFS	14	array out of bound access	5.16	fs/btrfs/struct-funcs.c:btrfs_get_16()	reported	O	O	X	O	O
	15	NULL pointer deref	5.17	fs/btrfs/ctree.c:btrfs_search_slot()	reported	O	O	X	O	O
	16	gen. Protection fault	5.16	fs/btrfs/struct-funcs.c:btrfs_get_32()	patched	O	O	X	O	O
	17	gen. Protection fault	5.17	fault at fs/btrfs/tree-checker.c: check_dir_item()	reported	O	O	X	O	O
	18	gen. Protection fault	5.17	fs/btrfs/print-tree.c: btrfs_print_leaf()	reported	O	O	X	O	O
	19	gen. Protection fault	5.17	fs/btrfs/treelog.c: btrfs_check_ref_name_override()	reported	O	O	X	O	O
	20	gen. Protection fault	5.18	fs/btrfs/file-item.c: btrfs_csum_file_blocks()	reported	O	O	X	O	O
	21	gen. Protection fault	5.15.57	fs/btrfs/volumes.c: btrfs_get_io_geometry()	reported	X	O	X	X	X
	22	gen. Protection fault	5.15.57	fs/btrfs/lzo.c: lzo_decompress_bio()	reported	X	O	X	X	X
	23	BUG()	5.19	fs/btrfs/inode.c: btrfs_finish_ordered_io()	reported	O	O	X	O	O
	24	BUG()	5.18	fs/btrfs/extent_io.c: extent_io_tree_panic()	reported	X	O	X	X	X
	25	BUG()	5.15.57	fs/btrfs/extent-tree.c: update_inline_extent_backref()	reported	X	O	X	X	X
	26	BUG()	5.15.57	fs/btrfs/root-tree.c: btrfs_del_root()	reported	X	O	X	X	X
	27	BUG()	5.18	fs/btrfs/delayed-ref.c: update_existing_head_ref()	reported	X	O	X	X	X
fs	28	BUG()		fs/inode.c:611	reported	O	O	X	O	O
F2FS	29	NULL pointer deref	5.15	CVE-2021–44879	patched	X	O	X	O	O
	30	use after free	5.15	CVE-2021–45469	patched	O	O	X	O	O
	31	array-index-out-of-bounds	5.17-rc6	fs/f2fs/segment.c:3460	patched	O	O	X	O	O
	32	NULL pointer deref	5.17	f2fs/dir.c:f2fs_add_regular_entry()	patched	O	O	X	O	O
	33	use after free	5.19	fs/f2fs/segment.c: f2fs_update_meta_page()	patched	O	O	X	O	O
	34	use-after-free	5.19	fs/f2fs/recovery.c:check_index_in_prev_nodes()	patched	X	O	X	X	X
	35	slab-out-of-bounds	5.15–6.0	fs/f2fs/segment.c:reset_curseg	reported	X	O	X	X	X

References

1. Aota, N., Kono, K.: File systems are hard to test—learning from XFStests. IEICE Trans. Inf. Syst. **102**(2), 269–279 (2019)
2. Corina, J., et al.: DIFUZE: interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2017)
3. Federal Communications Commission: What is 'Juice Jacking' and Tips to Avoid It, Federal Communications Commission (2023)

4. Jeong, D.R., Kim, K., Shivakumar, B., Lee, B., Shin, I.: Razzer: finding kernel race bugs through fuzzing. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy (2019)
5. Jeon, Y., Han, W., Burow, N., Payer, M.: FuZZan: efficient sanitizer metadata design for fuzzing. In: Proceedings of the 2020 USENIX Annual Technical Conference (ATC) (2020)
6. Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee. B.: HFL: hybrid fuzzing on the Linux kernel. In: Proceedings 2020 Network and Distributed System Security Symposium (2020)
7. Langner, R.: Stuxnet: dissecting a cyberwarfare weapon. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy (2011)
8. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of 2004 International Symposium on Code Generation and Optimization (CGO) (2004)
9. Lee, C., Sim, D., Hwang, J.Y., Cho, S.: F2FS: a new file system for flash storage. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST) (2015)
10. https://lore.kernel.org/all/20220704142721.157985-1-lczerner@redhat.com/
11. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices **40**(6), 190–200 (2005)
12. Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L.: The new ext4 filesystem: current status and future plans. In: Proceedings of the Linux Symposium (2007)
13. MITRE Corporation. CVE-2009–1235 (2009)
14. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44879
15. Mohan, J., Martinez, A., Ponnapalli, S., Raju, P., Chidambaram, V.: CrashMonkey and ACE: systematically testing file-system crash consistency. ACM Trans. Storage **15**(2), 1–34 (2019). https://doi.org/10.1145/3320275
16. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices **42**(6), 89–100 (2007)
17. Nossum, V., Casasnovas, Q.: Filesystem fuzzing with american fuzzy lop. In: Proceedings of Vault Linux Storage and Filesystems Conference (2016)
18. Pailoor, S., Aday, A., Jana, S.: MoonShine: optimizing OS fuzzer seed selection with trace distillation. In: Proceedings of the 27th USENIX Security Symposium (2018)
19. Peng, H., Payer, M.: USBFuzz: a framework for fuzzing USB drivers by device emulation. In: Proceedings of the 29th USENIX Security Symposium, USENIX Security (2020)
20. Purdila, O., Grijincu, L.A., Tapus, N.: LKL: the Linux kernel library. In: Proceedings of the 9th RoEduNet IEEE International Conference (2010)
21. Rodeh, O., Bacik, J., Mason, C.: BTRFS: the Linux B-tree filesystem. ACM Trans. Storage **9**(3), 1–32 (2013)
22. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kAFL: hardware-assisted feedback fuzzing for OS kernels. In: Proceedings of the 26th USENIX Security Symposium (2017)
23. SGI, OSDL and Bull: Linux Test Project (2023). https://github.com/linux-test-project/ltp
24. Song, D., et al.: PeriScope: an effective probing and fuzzing framework for the hardware-OS boundary. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS) (2019)
25. syzbot: Google (2023). https://Syzkaller.appspot.com/upstream
26. Syzkaller: Google (2023). https://github.com/google/Syzkaller
27. Syzkaller: Syscall descriptions (2022). https://github.com/google/Syzkaller/blob/master/docs/syscall_descriptions.md
28. Wang, D., Zhang, Z., Zhang, H., Qian, Z., Krishnamurthy, S.V., Abu-Ghazaleh, N.: Beating kernel fuzzing odds with reinforcement learning. In: Proceedings of the 30th USENIX Security Symposium (2021)

29. Wen, C., et al.: MemLock: memory usage guided fuzzing. In: Proceedings of the 42nd International Conference on Software Engineering (2020)
30. Xu, W., Moon, H., Kashyap, S., Tseng, P.N., Kim, T.: Fuzzing file systems via two-dimensional input space exploration. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP) (2019)
31. Xu, M., Kashyap, S., Zhao, H., Kim, T.: Krace: Data race fuzzing for kernel file systems. In: Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP) (2020)
32. Zalewski, M.: American Fuzzy Lop (2.52b) (2018). http://lcamtuf.coredump.cx/afl
33. Zhao, B., et al.: StateFuzz: system call-based state-aware linux driver fuzzing. In: Proceedings of the 31st USENIX Security Symposium (2022)