LFUZZ: Exploiting Locality for File-system Fuzzing

Wenqing Liu, *Florida State University* An-I Andy Wang, *Florida State University*

Abstract

Fuzzing, or program testing using randomized inputs, is a useful technique to detect bugs elusive to regression suites and human testing. The idea of using randomized inputs is to have a broad uniform reach of the code branches to increase the program test coverage.

When fuzzing a file system, sequences of file operations are one source of input. Its persistent file-system image is another source of input, as file operations retrieve from and store to the image. One interesting observation is that leading file-system fuzzers tend to permute operations and their parameters, accessing a *small* set of files to encourage the exploration of deep code branches; thus, the accessed file-system image locations may show decent locality. That also means fuzzing random file-system image locations is not as effective, as fuzzed file operations are unlikely to reference random image locations.

Another challenge is the minimum file-system image size a fuzzer has to track is large. Therefore, leading file-system fuzzers avoid saving and restoring modified images at the cost of, at times, regenerating system states and reducing the reproducibility of bugs.

We introduce LFUZZ, a file-system fuzzing framework that exploits the locality shown in typical fuzzing workloads. LFUZZ tracks recently accessed image locations and nearby locations to predict the locations that will be referenced in the near future. Our scheme is dynamic and adaptive to migrating file access patterns. Also, since modified image locations are localized, LFUZZ incrementally tracks file-system image changes, so that states can be fuzzed from intermediary images instead of from the top-level seed images. LFUZZ further explores the use of partially updated images to simulate missing writes.

We applied LFUZZ to fuzz ext4, BTRFS, and F2FS, and LFUZZ has found 17 new bugs. Compared to JANUS, LFUZZ reduced the fuzzing area by up to a factor of eight and increased the code execution branch coverage by up to 18%.

1. Introduction

File systems are perhaps one of the most important operating system components, as they hold consistent and persistent states to survive reboots and crashes. Bugs in file systems can potentially lead to bad consequences, ranging from deadlocking and crashing the operating system to losing data and exposing security vulnerabilities. An adversary can lure a user to mount a crafted file system image [Langner 2011] or issue a sequence of file operations that leads to vulnerabilities or escalation of privileges [MITRE 2009].

Traditional ways to eliminate file-system bugs heavily rely on manual testing and regression test suites [Atoa and Kono 2019; SGI 2022]. However, human enumerations of testing cases may miss bug triggers that involve complex sets of constraints to be met. It is also possible to exhaustively and systematically test workloads within a bounded space [Mohan et al. 2019] such as, for example, within a dozen file operations. The drawback of this approach is that it is missing bugs that involve more file operations.

One alternative is fuzzing, which uses accumulated random inputs and can find corner cases slipped from regression test suites. Syzkaller [2022] is perhaps the most well-known kernel fuzzer. With continuous fuzzing of syzbot [2022], it has reported 2,800+ bugs in 2.5 years to upstream Linux kernels. Other general kernel fuzzers include kAFL [Schumilo et al. 2017] and the fuzzers based on Syskaller [Wang et al. 2021]; all find a decent number of new bugs within days of fuzzing, indicating that fuzzing is promising to explore hard corner-case bugs.

When fuzzing file systems, the traditional notion of fuzzing is not sufficient. The first challenge is that a file system has two sources of inputs, sequences of file operations and persistently stored images. For a typical fuzzer, while file operations and parameters are randomly permuted, randomly fuzzing file-system images is not as effective. For example, fuzzing the content of i-node X will not affect the file-system execution coverage if the file operations only reference i-node Y.

A second challenge is that the minimum file-system size ranges from 8MB to 128MB. While it may seem small compared to the size of modern storage, if each fuzzing test involves saving and restoring a file-system image, both the performance overhead and storage capacity overhead can be prohibitive.

We introduce LFUZZ, a file-system fuzzing framework, to address these two challenges. The key observation is that while file operations and parameters are permuted during fuzzing, typically, only a small set of files (i.e., <100 within 240 CPU fuzzing hours) are accessed to encourage deeper state explorations, even for a file system prepopulated with many files. This means that the referenced file system image locations may show a decent locality. Thus, by fuzzing the image locations likely to be referenced next, we can reduce the number of fuzzing iterations that yield no new execution coverage. The locality of image updates also means smaller and clustered modified image ranges, allowing us to save and restore file system images in the form of deltas. Additionally, we discovered that incompletely restoring deltas simulates missing writes, another form of testing that leads to many bug discoveries.

We applied LFUZZ on ext4 [Mathur et al. 2007], BTRFS [Rodeh et al. 2013], and F2FS [Lee et al. 2015] for 240 CPU hours. Compared to JANUS, LFUZZ increased the code execution branch coverage by up to 18% [Xu et al. 2019]. Furthermore, LFUZZ discovered 17 new bugs.

In summary, we have made the following contributions to this work:

- We have identified that file-system image fuzzing is insufficient because many fuzzed locations are not referenced by file operations.
- We analyzed the locality feature of fuzzing file system workloads on file-system image modifications and proposed the locality-aware fuzzing approach for kernel file systems.
- We applied image deltas with missing write simulations to find file-system bugs.
- We designed, implemented, and evaluated the LFUZZ prototype, which increased the branch coverage by up to 18% compared to JANUS, and found 17 new bugs.

The remaining paper is structured as follows. Section 2 shows the limitations of leading file-system fuzzers. Section 3 examines image reference locality under leading fuzzing workloads. Sections 4, 5, and 6 present our LFUZZ design, implementation, and evaluation. Section 7 relates our work with existing fuzzers. Section 8 discusses this study's limitations and directions for future work, and Section 9 concludes the paper.

2. Leading File-System Fuzzers

Since saving and restoring file-system images are expensive, the designs of leading file-system fuzzers try to avoid this cost.

Syzkaller [2022]: for image fuzzing, Syzkaller first creates a file-system image by picking an `mkfs` parameter set and prepopulateing the file system. Then, a random sequence of fuzzed file operations is applied. File operation fuzzing is similar, but file operations are tested one after another without resetting the kernels until the container VM reaches the time limit or needs to reboot. Note that the notion of randomness here is constrained by the file system semantics. A write system call can only be issued to a file that is already open [Syzkaller 2022a]. This constraint limits the number of files being fuzzed in one execution (within 240 CPU fuzzing hours, we see the maximum referenced file number is 4, and the maximum number of operations on files is 13, but the average number of operations on files is only 2).

One consequence of not resetting file system images between sequences of file operations is that when a bug is detected (e.g., system crashes, kernel panics, BUG() error message, KASAN [Jeon et al. 2020] error messages, time outs), it is difficult to discern whether it is caused by the latest sequence of file operations or the cumulative changes of system states leading to this point. When the latest sequence of file operations is applied to the original image, the bug reproducibility rate is only around 50% based on our experience. Xu et al. [2019] found that all crash-related bugs for Syzkaller are not reproducible. Additionally, Syzkaller does not fuzz the file system images.

AFL [Zalewski 2018], a popular fuzzing tool based on genetic algorithms, has also been used to fuzz file-system images [Nossum and Casanovas 2016]. The fuzzed images can be mounted to run regression test suites. AFL narrows down the file-system image by fuzzing only nonzero metadata blocks. Corrupted data blocks generally pose little threat to file-system integrity; thus, they are omitted for fuzzing. An unintended side effect is that AFL may skip valid metadata blocks that are zero-initialized. Another issue is that for copy-on-write file systems, new versions of metadata are written elsewhere instead of updating metadata in place, littering obsolete nonzero metadata blocks behind, diluting the fuzzing targets. Fuzzing obsolete metadata blocks would not contribute to finding new execution branches.

JANUS [Xu et al. 2019] is built on a variant of the AFL fuzzing code base, and it fuzzes both file operations and file system images. To increase the chance of fuzzing image locations that will be referenced, JANUS extracts the initial metadata regions from a given file-system image with prepopulated files, and it would fuzz only the fixed metadata region.

A JANUS fuzzing ***round*** starts with an image fuzzing phase, with a selected file system image, say I_0. JANUS then fuzzes a metadata location to create I_1, and applies a random file operation F_0 (complying with file-system semantics). A new execution coverage is detected when a new transition is found between two compiled basic blocks. If no new execution coverage is found, JANUS returns to I_0, starts a new ***iteration***, fuzzes another metadata location to create I_2, and reapplies F_0 to I_2. If a new execution coverage is found, JANUS saves the metadata region of I_2 and F_0, along with file states indicating whether a file is open, etc.

After enough iterations (depending on the coverage found so far with I_0), if JANUS cannot find any coverage during the image fuzzing phase, it enters the second phase of file-operation fuzzing (still within the same round). Since we have discovered new coverage, the file operation phase is skipped in this case. The saved image with the highest priority is chosen based on the AFL seed scheduling scheme. In this case, I_2 is chosen for the next round.

However, suppose no coverage is found during the first phase. Random file operations are selected with mutated arguments or appended to the file operation sequence. After each mutation or appending, file operations are applied to the original unfuzzed I_0. After enough iterations (depending on the coverage found so far), the saved image with the highest coverage increase is chosen.

One detail to handle with image fuzzing is checksummed blocks (e.g., superblocks). Fuzzing a superblock will likely lead to mount failures, which precludes the exploration of deeper code branches behind the checksum verification. Thus, JANUS will fix various checksums to be consistent with fuzzed content, simulating corruptions that occur immediately before checksums are computed.

Even though JANUS has narrowed down the fixed initial metadata region for fuzzing, the range of metadata regions is still large, while most fuzzing rounds focus on <100 files within 240 CPU hours. Also, JANUS will not fuzz dynamically allocated metadata blocks located beyond the initial metadata regions.

3. Image Reference Locality of FS Fuzzers

Based on leading file-system fuzzers, our intuition is that fuzzing references to a file-system image is far from random. Thus, we examined the size of referenced areas for a fuzzing iteration, their temporal relationship across fuzzing iterations, and their interactions with structured file-system layouts.

3.1. Size of Referenced File-system Image Locations

By intercepting `bio_endio()`, we traced the referenced locations on a file-system image for a sequence of 200 random file operations applied under JANUS. The results for ext4, BTRFS, and F2FS are tabulated in Table 3.1. The total referenced image size for a given sequence of file operations is only up to 0.02% of the smallest file system image, reflecting that JANUS fuzzing focuses on metadata. Although JANUS has narrowed down the fuzzing range to the initial metadata region, the actual referenced image size is still only up to 13% of the initial metadata size. That means if we randomly fuzz an image location, the chance of the fuzzed region being referenced is small. Another implication is that if we want to frequently track, save, and restore just the modified file-system image locations, the overhead may be affordable.

Table 3.1. The size of referenced image locations for 200 random file operations under JANUS.

	ext4	BTRFS	F2FS
Smallest file system image	8MB	128MB	64MB
Initial metadata size fuzzed under JANUS	111KB	41KB	90KB
Accessed image size	1.3KB	3.3KB	12KB
Percentage of file-system image bytes referenced	0.02%	0.003%	0.02%
Percentage of initial metadata bytes referenced	1%	8%	13%

3.2. Temporal correlations of referenced image locations

Another question is how well the currently fuzzed image locations correlate with the next iteration of fuzzed image locations. For JANUS, the same file operation sequence is applied to many fuzzed file-system images during the image fuzzing phase before the next mutated file operation is appended to the sequence. Thus, it is highly likely that the image reference locations of one iteration are correlated with the next.

We traced referenced image locations for the JANUS fuzzing for 6,000 iterations and found that for ext4, 78% of referenced image locations for one iteration overlap with the referenced image locations of the next iteration. Similarly, for BTRFS, the overlapping rate is 75%; for F2FS, 80%. Thus, by fuzzing the current referenced image locations, we can have a high chance of being referenced by the next iteration of file operations.

3.3. Spatial Correlations of Referenced Image Locations

Since file system images are highly structured and metadata blocks are allocated systematically, we examined whether there is a distance relationship between the updated blocks from one iteration to the next iteration. Suppose iteration one updates blocks 1 and 2; iteration two updates blocks 2, 3, and 4. We compute all pair-wise distances from newly referenced blocks from the second iteration to blocks from the first iteration: (3 – 1), (3 – 2), (4 – 1), (4 – 2). Thus, we will have 2, 1, 3, and 2. So the newly referenced block has a 50% chance of being 2 blocks away from any blocks in the first iteration and a 25% chance of being 1 or 3 blocks away. We bound the distance to 50 blocks. Since an update may involve different metadata structures located in different areas (e.g., journal and i-node blocks), the distance between these areas little reflects how metadata blocks of the same type are allocated.

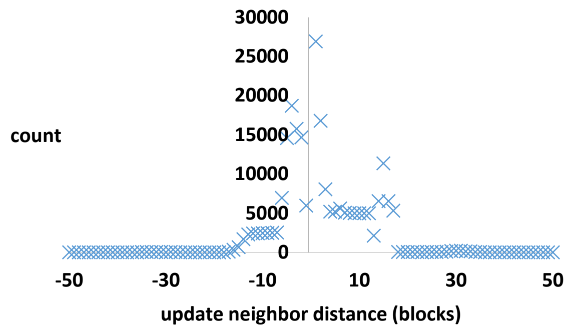

Figure 3.3.1: Frequency distribution of distances between updated blocks between iterations for ext4.

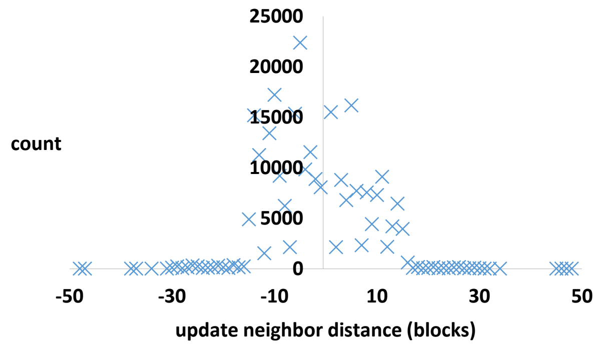

Figure 3.3.2: Frequency distribution of distances between updated blocks between iterations for BTRFS.

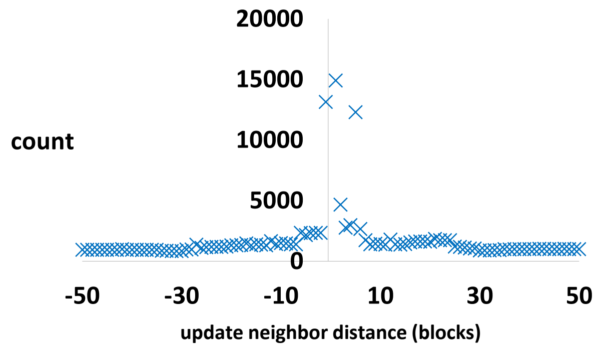

Figure 3.3.3: Frequency distribution of distances between updated blocks between iterations for F2FS.

We ran JANUS for two hours to perform data gathering. Figures 3.3.1-3.3.3 present the results. For ext4, the most popular update neighbor distance is 1, reflecting that blocks are linearly and incrementally allocated. Also, when fuzzing the next iteration, the next referenced blocks are likely to be within three blocks of a block referenced within the current iteration. For BTRFS, the range is more scattered due to the use of b-trees. For F2FS, the most popular update neighbor distances are 1, -1, and 5. Thus, fuzzing blocks surrounding referenced blocks for the current fuzzing iteration can increase the chance of them being referenced by the next sequence of file operations.

4. LFUZZ Design

Although the concept of locality is applied extensively to optimize storage systems, applying locality to enhance fuzzing is counterintuitive since fuzzing thrives on permuting random inputs to broaden the code execution branch coverage. Based on the evidence we found in Section 3, we introduce LFUZZ, a file-system fuzzing framework that exploits both spatial and temporal localities when fuzzing file-system images. For temporal locality, LFUZZ fuzzes currently referenced image locations, since the next fuzzing iteration will likely have overlapping referenced locations. For spatial locality, LFUZZ also fuzzes neighboring blocks of currently referenced image locations, since the next fuzzing iteration is likely to reference surrounding locations.

LFUZZ maps file-system image locations into cached memory locations. By intercepting memory references made by a file system, LFUZZ can track referenced file-system image locations at a fine granularity. Referenced file-system image locations are stored in our least-recently-used (LRU) list with a bounded length to adapt to locality changes over time. Thus, for each iteration, LFUZZ will fuzz these file-system referenced locations plus some neighboring locations.

One implication of this list is that when we save a file system image because of the discovery of new coverage, we also need to save this LRU list. So, when the saved image is restored for further fuzzing, the locality information is also restored. Each list element size is only a fraction of a storage block (i.e., 64B) to reduce the storage and saving/restoration overheads.

To reduce the cost of saving file system images, we also introduced the notion of deltas, which can be obtained by subtracting the modified image I' from image I before applying fuzzed the file operation. Again, we use a sub-block granularity (e.g., 256B) to reduce the storage and the saving/restoration overhead.

Having deltas can tame the algorithmic complexity of fuzzing. In JANUS's file-operation fuzzing phase, to avoid saving/restoring images, an original image is fuzzed with an increasing length of file operations in an O(n^2) manner since iteration appends one new file operation, and all preceding operations need to be reapplied. LFUZZ delta fuzzing only needs to apply one new file operation to the delta image accumulated in each iteration. In addition, we discovered that partially restored deltas could lead to many file-system bugs; thus, we incorporated this technique into our delta fuzzing.

We leveraged the Linux Kernel Library [Purdila et al. 2010] to fuzz file systems in the user space to avoid expensive system reboots. The system rebooting cost is replaced with resetting the file system image. Also, all fuzzing states and file system images are memory-resident via the use of shared memory. To avoid reinventing the wheel, we leveraged the fuzzing infrastructure of JANUS, which has file operation permutation mechanisms that comply with file-system semantics. JANUS also has built-in genetic algorithms from AFL that can be used to fuzz targeted file-system image areas.

4.1. Tracking Referenced Image Locations

LFUZZ intercepts requests with stubs inserted at the block layer to create a file-system image-location-to-memory-address mapping (F2M) table (Table 4.1.1). LFUZZ instrumented the stubs to report referenced memory addresses. A referenced file-system image location can be identified whenever a referenced memory address is found in the F2M table. For example, if memory address 0x7FF0041a3008 is referenced, it is within 0x7FF0041a3000 + page size (0x1000 bytes); thus, file-system image block 9217 is referenced.

Table 4.1.1: File Image Location to Memory Address Table.

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

Once the block number is identified, it is stored in our LRU list. Since LRU-tracked content regions need to be saved and restored along with images with newly found coverage, we used a sub-block granularity of 64B, or a bucket to reduce the storage and saving/restoration overhead. In Table 4.1.2, each LRU list item tracks a bucket. In this case, this reference lands in bucket 0 of image block 9217.

Table 4.1.2: LRU List.

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

4.2. Exploiting Locality for Fuzzing

During various fuzzing phases, the referenced image locations can change after certain bytes are mutated. To increase the reference rate after a fuzzing iteration, we exploit spatial and temporal localities of file system behaviors.

4.2.1. Temporal Locality

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

The LRU list is also bounded, so that less frequently accessed locations will be dropped as potential fuzzing targets. For copy-on-write file systems, for example, updated metadata blocks are written elsewhere instead of in-place, leaving obsolete metadata blocks behind, and diluting the quality of potential fuzzing targets. Dropping them from the fuzzing targets increases the chance of fuzzed locations being referenced in the next iteration.

4.2.2. Spatial Locality

The referenced image location changes when metadata areas are fuzzed. For instance, when `open` is called, a file system needs to assign an unused i-node to the created file. If the i-node bitmap is fuzzed in a way such that some unused i-nodes are marked as used, then the new i-node will be allocated to skip entries that are marked as used. To increase the chance of newly fuzzed image locations being referenced in the next fuzz iteration, we chose neighboring locations and referenced locations as fuzzing targets.

Intra-block locality: Since image fuzzing typically starts from a sparse state, with a limited number of prepopulated files, i-nodes, directory entries, etc. tend to be allocated in succession. This means, that when an allocation is requested, it is highly likely that the system will pick the free space near the currently used ones. We used a bucket size of 64B instead of the actual bytes referenced. Our measurements report that 78% of image bucket locations referenced in this fuzz iteration are referenced in the next iteration for ext4, 75% for BTRFS, and 80% for F2FS. Thus, by fuzzing the currently referenced image locations and surrounding locations, these locations are likely to be referenced in the next fuzz iteration.

Inter-block locality: Since metadata block allocations can be allocated in succession, neighboring blocks are likely to be referenced in the next fuzzing iteration as well. Another possible scenario is for copy-on-write file systems, an updated metadata block in memory may be written to another (potentially neighboring) metadata block on storage. Thus, if we reference a bucket offset within the current block, we would add the same bucket offset of neighboring blocks as potential targets for fuzzing.

4.3. Image Deltas

Given the locality in fuzzing workloads, we devised the use of image deltas to reduce the file-system image storage and saving/restoration overheads. An image delta D is defined as the modified file-system image I', subtracting the original image I_0 before modifications. This subtraction can be expensive if only a few places are modified due to locality. Thus, we applied a copy-on-write mechanism on I_0, so that only the modified image regions are copied and tracked.

Unlike leading file-system fuzzers where images are saved when new coverage or bugs are found, delta images are sufficiently lightweight that can be saved whenever a file-system image is modified. That means 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 file system state changes for all proceeding file operations. Thus, during the file-operation fuzzing phase of JANUS, we can reduce the $O(n^2)$ file operations applied down to $O(n)$, albeit, the overhead per iteration is higher due to the need to save and restore delta images.

4.4. Missing Writes

While developing our image delta technique, we discovered that partially restored deltas led to file-system bugs and crashes. Further investigation revealed that partially restored deltas simulate missing writes, where two versions of file system states are co-mingled. Thus, segments of the file system states are self-consistent, while globally, the file system states are inconsistent. Since we discovered quite a few bugs this way, we incorporated this fuzzing technique into delta fuzing. The probability of triggering a missing write is the current length of system call sequence L, divided by (L + 5). This means that when the system call sequence is short, LFUZZ is likelier to use delta fuzzing. When the system call sequences grows longer, LFUZZ is likelier to fuzz using missing writes.

4.5. LFUZZ Phases

LFUZZ has the following fuzzing orders in three phases—LRU-based fuzzing, JANUS-based system-call fuzzing, and system-call fuzzing with delta. The fuzzing ordering is similar to JANUS's fuzzing order, to ease our comparison. Also, similar to JANUS, the next phase is only triggered, when the current phase cannot find any new coverage.

Figure 4.5.1 presents the LRU-based fuzzing phase, where a single file operation is applied to different fuzzed images. At line 1, a new corpus is loaded with an initial image I, a sequence of file operations F, and LRU regions L from the first fuzzing iteration. The LRU regions L are first fuzzed (line 2), and the fuzzed regions L' are distributed to the original image I (line 4) to build a fuzzed image I'. A sequence of file operations F is applied to the fuzzed image I' (line 5). If new coverage is found, save modified image I', the file operation sequence F, and modified LRU regions L' (line 7). The number of fuzzing iterations is adaptive depending on whether new coverage is found (lines 8-9). At the end of the iteration, LRU regions L' are fuzzed for the next iteration (line 12).

```
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  }
```

Figure 4.5.1: LFUZZ LRU-based fuzzing phase.

```
1   for corpus C = {image I, file ops F, image delta D} {
2     F' = F + file op // append a new file op
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
8       } else {
9         save(I, F', D')
10        if  (B < max_bound) {
11          B *= 2;
12        }
13        F' += fuzzed file op // append a new file op
14      }
15    }
16  }
```

Figure 4.5.2: LFUZZ delta-image-based fuzzing phase.

Figure 4.5.2 presents the delta-based system-call fuzzing phase, where an incrementally increased file operation sequence is applied to delta images updated after each iteration. At line 1, a new corpus is loaded with an initial image I, a sequence of file operations F, and a saved delta image D. The sequence of file operation F is appended with a newly selected file operation to form F' (line 2). A fuzzed image D' is formed by applying delta regions D to the initial image I (line 3). The newly formed delta image D' is updated by applying the new file operation sequence F' to itself to accumulate states for each iteration (line 5). If no new coverage is found, move on to the next corpus (line 7). If new coverage is found, save the initial image I, the updated file operation sequence F', and the updated delta image D' (line

9) and increase the number of iterations adaptively (lines 10-11). Finally, append a new file operation to F' for the next iteration (line 13).

5. Implementation

The development tool-chain eco-system of a file-system fuzzing framework is complex, which made it difficult to start our framework from scratch. We choose JANUS over Syzkaller as a starting point in favor of the reproducibility of crash-based bugs. Since JANUS is based on the Linux Kernel Library, which is not frequently updated, we had to port newer versions of the Linux kernel.

We need to add wrapper functions for each file system, to fix checksums and distribute the fuzzed contents from LRU regions to the JANUS fuzzing buffer.

Tracking file-system reference locations contributes most of the system overhead, as it traces all memory references and extracts the ones that access cached file-system image pages. If the overhead is too high, it can neutralize the benefit of locality-based image fuzzing. So, instrumentation tools such as Intel PIN [Luk et al. 2005] and Valgrain [Nethercote and Seward 2007] are not suitable as they incur too much overhead. Instead, we used LLVM [Lattner and Adve 2004] to inject instrumentation at compile time. With no need to run an emulator at run time, the overhead is significantly lower when compared to Intel PIN.

For delta fuzzing, to track incremental updates, we leverage the JANUS's page-fault handler in `userfaultfd` to compare locations of file-system images that triggered page faults. The delta is obtained at the end of a page fault execution to compare page fault locations of volatile memory with the cached file-system image memory addresses.

Table 5.1 summarizes the line count for the LFUZZ implementation.

Table 5.1: LFUZZ implementation line counts

LLVM runtime (+LRU fuzzing)	1,193
LLVM pass	239
File-system wrapper	864
AFL	264
`bio_stub`	64
Delta fuzzing	72

6. Evaluation

We tested LFUZZ in a VM on a Dell Precision 7820 with Intel® Xeon® Gold 5218R 40 cores with 128GB of memory. The tests of LFUZZ and JANUS are performed with 10 processes each. The figures are presented with a 90% confidence interval, unless otherwise specified. Since JANUS has already demonstrated more effective coverage than Syzkaller [Xu et al. 2019], we will only compare with JANUS. To evaluate how LFUZZ performs, we focused on the following questions:

- How well can LFUZZ find new bugs?
- How well can LFUZZ increase coverage?
- How well LFUZZ can narrow down the fuzzing range?

6.1. New Bugs

We ran LFUZZ and JANUS for a week and found 30 new bugs (Table 6.1.1), and 17 of them were only found by LFUZZ. Among the unique bugs, 10 of them are memory bugs that have security implications. The bugs were reported to either Red Hat or upstream maintainers. Six of them are patched and three requested CVE numbers are assigned. LFUZZ found more bugs in ext4 and BTRFS, which have more features. F2FS with less complicated features is well fuzzed by JANUS. Most of the unfuzzed code regions are from `ioctl`-related code, which is not fuzzed by JANUS.

6.1.1. Case Study: CVE-2022-A

JANUS cannot find this bug because it lies in `do_split()`. The function is executed when an image is almost filled-up with prepopulated files. However, after JANUS extracts the metadata range, the fuzz area is too large for its AFL component to fuzz. That means, reducing candidate fuzzing area matters, and exploiting locality helps. LFUZZ can fuzz smaller file-system image areas to trigger this bug.

6.1.2. Case Study: CVE-2021-B

The cause of this bug is that the fuzzed image makes a data block a special file (e.g., character, block, FIFO, or socket file). When the block is to be migrated due to F2FS garbage collection, it calls `a_ops->set_dirty_page()`, but the operation pointer is `NULL` for the special files, triggering a `NULL` pointer dereference.

To trigger this bug, the fuzzer needs 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. Meanwhile, if the fuzzed i-node `imode` or SSA entries are in the state to trigger the bug, the block has to be migrated to make it happen.

JANUS fuzzes the initial metadata block locations. For F2FS, the total size is 90 KB. During the first iteration of fuzzing, LFUZZ tracked 12KB as potential fuzzing locations, which is about one-seventh of JANUS. Focused image fuzzing helped us find this bug. Syzkaller generated a bug report with a similar call stack but did not provide a reproducer, making it hard to find the root cause.

Table 6.1.1: Bugs found by LFUZZ and JANUS.

File systems	Bug type	Affected version	Bug	Status	Found by JANUS	Found by LFUZZ
ext4	Stack-out-of-bounds	5.18	__blk_flush_plug	acknowledged	X	O
	use after free	5.18	fs/ext4/namei.c: do_split()	acknowledged	X	O
	out-of-bounds read	4.19	ext4_search_dir()	patched	X	O
	use after free	5.18	fs/ext4/namei.c:dx_insert_block() CVE-2022-A	confirmed	X	O
	Slab-out-of-bounds	5.18	fs/ext4/xattr.c: ext4_xattr_set_entry()	reported	X	O
	use after free	5.18	fs/ext4/namei.c:ext4_insert_dentry()	reported	X	O
	BUG()	5.18	fs/ext4/extents_status.c:202	reported	O	O
	BUG()	5.18	fs/ext4/ext4_jbd2.h:ext4_inode_journal_mode()	reported	X	O
	BUG()	5.18	fs/ext4/extent.c:ext4_ext_determine_hole()	patched	X	O
BTRFS	array out of bound access	5.16	fs/btrfs/struct-funcs.c:btrfs_get_16()	patched	O	O
	NULL pointer dereference	5.17	fs/btrfs/ctree.c:btrfs_search_slot()	reported	O	O
	general protection fault	5.16	fs/btrfs/struct-funcs.c:btrfs_get_32()	patched	O	O
	general protection fault	5.17	fault at fs/btrfs/tree-checker.c: check_dir_item()	reported	O	O
	general protection fault	5.17	fs/btrfs/print-tree.c: btrfs_print_leaf()	reported	O	O
	general protection fault	5.17	fs/btrfs/treelog.c: btrfs_check_ref_name_override()	reported	O	O
	general protection fault	5.18	fs/btrfs/file-item.c: btrfs_csum_file_blocks()	reported	O	O
	general protection fault	5.15.57	fs/btrfs/volumes.c: btrfs_get_io_geometry()	reported	X	O
	general protection fault	5.15.57	fs/btrfs/lzo.c: lzo_decompress_bio()	reported	X	O
	BUG()	5.19	fs/btrfs/inode.c: btrfs_finish_ordered_io()	reported	X	O
	BUG()	5.18	fs/btrfs/extent_io.c: extent_io_tree_panic()	reported	X	O
	BUG()	5.15.57	fs/btrfs/extent-tree.c: update_inline_extent_backref()	reported	X	O
	BUG()	5.15.57	fs/btrfs/root-tree.c: btrfs_del_root()	reported	X	O

	BUG()	5.18	fs/btrfs/delayed-ref.c: update_existing_head_ref()	reported	X	O
fs	BUG()	5.18	fs/inode.c:611	reported	O	O
F2FS	NULL pointer dereference	5.15	CVE-2021-B	patched	X	O
	use after free	5.15	CVE-2021-C	patched	O	O
	array-index-out-of-bounds	5.17-rc6	fs/f2fs/segment.c:3460	patched	O	O
	NULL pointer dereference	5.17	f2fs/dir.c:f2fs_add_regular_entry()	patched	O	O
	use after free	5.19	fs/f2fs/segment.c: f2fs_update_meta_page()	patched	O	O
	use after free	5.19	fs/f2fs/recovery.c:check_index_in_prev_nodes()	patched	X	O

6.2. LFUZZ Coverage

We fuzzed JANUS and LFUZZ under each configuration for 240 CPU hours and compared their code branch coverage, defined as unique edge transitions between compiled basic blocks. We ran LFUZZ with and without delta fuzzing. For the LRU fuzzing option, we tested list lengths of 512 and 2,048 buckets, which can hold 32KB and 128KB, respectively. Figures 6.2.1-6.2.3 present the edge coverage results for ext4, BTRFS, and F2FS.

Overall, the branch coverages under various LFUZZ configurations are comparable to JANUS. Note that with delta enabled, LFUZZ will have three fuzzing phases instead of JANUS's two phases, which could impose some overhead. In the best cases, LFUZZ edge coverage can outperform JANUS up to 18% for ext4, 6% for BTRFS, and 13% for F2FS.

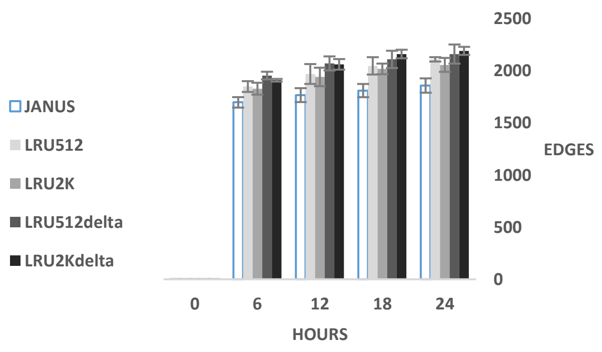

Figure 6.2.1: ext4 branch coverage comparison between JANUS and LFUZZ under different configurations.

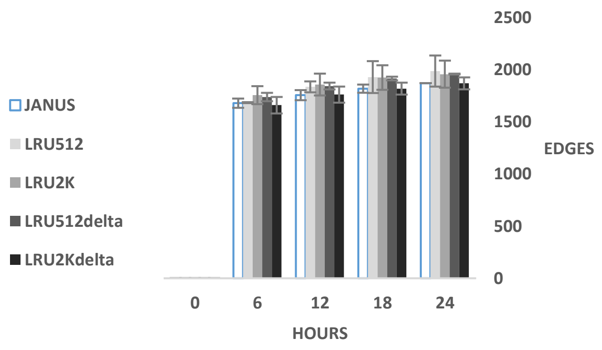

Figure 6.2.2: BTRFS branch coverage comparison between JANUS and LFUZZ under different configurations.

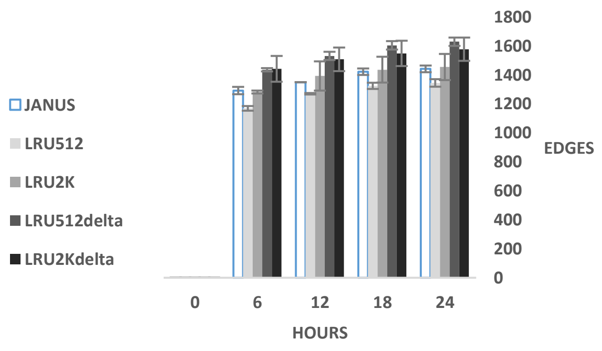

Figure 6.2.3: F2FS branch coverage comparison between JANUS and LFUZZ under different configurations.

One interesting observation is that not a single LFUZZ configuration can achieve the best coverage for all three file systems. For ext4 (Figure 6.2.1), since the working set was smaller than 512 buckets, the edge coverage was about the same with a longer LRU length bound. The combination of LRU fuzzing and delta fuzzing achieved the highest coverage. For BTRFS (Figure 6.2.2), LRU fuzzing performed better, since many referenced metadata items became obsolete due to copy-on-write, which diluted the fuzzing candidate area. However, delta fuzzing did not seem to contribute as much to the edge coverage. For F2FS (Figure 6.2.3), 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. On the other hand, delta fuzzing, when combined with LRU, increased the edge coverage. Out of curiosity, we tested delta fuzzing alone without LRU (not shown); the edge coverage is only about 1K.

Overall, we found that it was difficult to attribute the cause of the coverage increase to a particular fuzzing phase since new coverage could be built on previously discovered and saved coverage by going through different fuzzing phases.

6.3. LFUZZ Fuzzed Regions

Figure 6.3.1 compares the sizes of fuzzed regions between JANUS and LFUZZ under different configurations. The figure shows the regions being fuzzed during the 24th hour of experiments, with error bars indicating the variation within one standard deviation. For ext4, LFUZZ's fuzzed area can be one-eighth of JANUS'S fuzzed area while achieving an 18% increase in edge coverage. For BTRFS, LFUZZ's fuzzed area can be 30% smaller, while achieving a 6% increase in edge coverage. As for F2FS, if LFUZZ is configured with 2K LRU buckets, the fuzzed area can be as large as JANUS, reflecting F2FS's wear leveling for its designed use on SSD devices. However, when LRU fuzzing is combined with delta fuzzing, LFUZZ can still achieve a 13% increase in edge coverage with a fuzzed area that is only 34% as large as JANUS's.

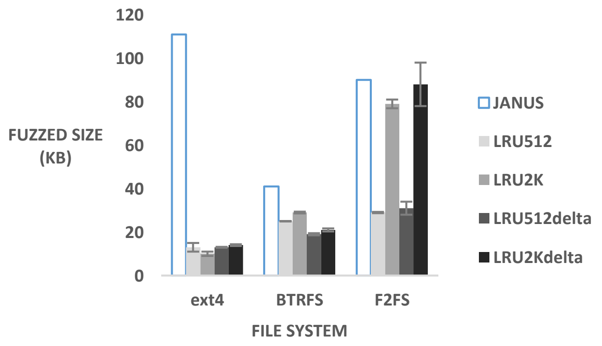

Figure 6.3.1: Comparison of fuzzed region sizes between JANUS and LFUZZ under different configurations.

We also found that the bound on the LRU length interacted with the fuzzing results. If the bound is too large, the content held by our LRU list approaches the entire working set, which contains both frequently and infrequently accessed areas for fuzzing. If the bound is too small, useful content is removed before it is referenced again in the near future. Due to the exponential explosion of the experimental space, we only systematically tested the lengths of 512 and 2048 buckets.

We also tested some extreme LRU length values. For example, on ext4, we tested an LRU length of 36 buckets, which can hold about 2KB of content. Since for each update operation, ext4 accesses the journal last, the LRU list mostly held journal content for fuzzing, with prior content removed due to the limit for LRU length. We were able to find a journal bug at `ext4_jbd2.h: ext4_inode_journal_mode()` after 12 hours. This bug did not appear in the first 12 hours when the LRU list length was longer than 64 buckets. Future work will focus on optimizing LRU list length.

7. Related Work

7.1. File-System Fuzzers and Exercisers

The closest work to LFUZZ is JANUS [Xu et al. 2019] and Syzkaller [2022]. JANUS fuzzes only the fixed initial metadata regions. Thus, JANUS will not fuzz these new regions if metadata regions migrate due to dynamic allocation or copy-on-write semantics. LFUZZ, on the other hand, uses fine-grained adaptive LRU tracking to track changes in metadata regions. The target fuzzing region of the file-system image can be one-eighth of the target region of JANUS; we can increase the code branch execution by up to 18%.

Syzkaller uses QEMU to fuzz the kernel and kernel coverage as feedback to guide the fuzzing. Syzkaller does not reset the file-system image across different system-call fuzzing sequences. Thus, bugs may not be reproduced by applying just the latest system call sequence on the initial image. The bug reproducibility rate is about 50%, and crashes are not reproducible [Xu et al. 2019]. LFUZZ offers delta-fuzzing that tracks modified images after each file operation is applied. The bug reproducibility rate is about 85%.

AFL [Nossum and Casanovas 2016] file-system fuzzer fuzzes nonzero image locations, which could exclude zero-initialized metadata and include obsolete metadata blocks littered by copy-on-write mechanisms.

CrashMonkey [Mohan et al. 2019] exhaustively tests file systems with a bounded input space (e.g., short file operation sequences). It constructs file-system crash states and runs file-system recovery operations. It then compares the file system states to detect bugs such as incorrect file sizes, files not removed during renames, etc. Although many bugs can be triggered within short file operation sequences, CrashMonkey can miss bugs caused by longer file operation sequences. Also, LFUZZ finds mostly memory bugs, including stack out-of-bounds, memory use-after free, array out-of-bound, general protection fault, NULL pointer dereferencing, BUG(), etc.

7.2. Kernel Fuzzers

Some of the kernel fuzzers focus on building an effective framework. KAFL [Schumilo et al. 2017] used the Intel® processor-tracer result to guide the fuzzing to reduce overhead. HFL [Kim et al. 2020] utilized symbolic execution to solve hard branches with complex logic. USBFuzz [Peng and Payer 2020] and Periscope [Song et al. 2019] fuzzed the drivers by modifying the MMIO and DMA interfaces.

Some kernel fuzzers focus on improving the quality of system-call sequences for fuzzing. When new coverage is detected, Moonshine [Pailoor et al. 2018] exploits system-call read/write dependencies to filter out calls that do not contribute to the state changes of the new coverage, thus minimizing the length of the system call sequence for further fuzzing. DIFUZE [Corina et al. 2017] analyzed `ioctl`-related code to generate valid structured input to fuzz drivers.

Instead of using code coverage as feedback, some improved the feedback strategy. SyzVegas [Wang et al. 2021] changed the way to schedule the seed images using the multi-armed-bandit algorithms. StateFuzz [Zhao et al. 2022] tracks variables that lead to state changes to prioritize test cases.

Razzer [Jeong et al. 2019] used the point-to information from static analysis to generate test cases likely to cause race conditions. Krace [Xu et al. 2020] used potential interleaving memory access instructions as coverage to guide the fuzzing to find race conditions.

8. Limitations and Future Work

Since the configuration space of LFUZZ is large, a systematic exploration would involve an exponentially large number of experiments. Thus, we did not conduct a fine-grained exploration of different LRU lengths and their effects on various file systems. 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 will conduct optimization studies for LFUZZ in the future. Since each file system interacts with LFUZZ very differently, we will also explore file-system-specific fuzzing.

Some code execution branches are controlled by compile time configuration, which means fuzzing itself can never reach some code regions. Features like big file handling cannot be covered without compiling the code with certain flags enabled. We will explore a different fuzzing framework for fuzz compiler-enabled code branches.

The file-system sizes we fuzzed were also small; thus, we were unable to fuzz code branches triggered by large file sizes (e.g., 500MB).

10. Conclusions

We have designed, implemented, and evaluated LFUZZ, a file-system fuzzer that exploits locality to enhance file-system image fuzzing. With our locality observation, we found it feasible to integrate delta file-system image tracking and enable incremental fuzzing instead of reapplying file operations from the top-level seed image. We found a new way to fuzz file systems with incompletely restored delta images to simulate missing writes. With all these techniques applied, LFUZZ has found 17 bugs. LFUZZ can also reduce the target fuzzing region by a factor of up to eight compared to JANUS and increase the code execution coverage by up to 18%.

References

[Atoa and Kono 2019] Aota N, Kono K. File Systems Are Hard to Test—Learning from XFStests. *IEICE Transactions on Information and Systems*, 102(2):269-279, 2019.

[Corina et al. 2017] Corina J, Machiry A, Salls C, Shoshitaishvili Y, Hao S, Kruegel K, Vigna G. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017,

[Jeon et al. 2020] Jeon Y, Han W, Burow N, Payer M FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[Jeong et al. 2019] Jeong DR, Kim K, Shivakumar B, Lee B, Shin I. Razzer: Finding Kernel Race Bugs through Fuzzing. *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019.

[Kim et al. 2020] Kim K, Jeong DR, Kim CH, Jang Y, Shin I, Lee B. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings 2020 Network and Distributed System Security Symposium*. 2020.

[Langner 2011] Langner R. Stuxnet: Dissecting a Cyberwarfare Weapon. *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[Lattner and Adve 2004] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of 2004 International Symposium on Code Generation and Optimization (CGO)*, 2004.

[Lee et al. 2015] Lee C, Sim D, Hwang JY, Cho S. F2FS: A New File System for Flash Storage. *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[Luk et al. 2005] Luk CK, Cohn R, Muth R, Patil H, Klauser, A, Lowney G, Wallace S, Reddi VJ, Hazelwood K, Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices*, 40(6):190-200, 2005.

[Mathur et al. 2007] Mathur A, Cao M, Bhattacharya S, Dilger A, Tomas A, Vivier L. The New ext4 Filesystem: Current Status and Future Plans. *Proceedings of the Linux Symposium*, 2007.

[MITRE 2009] MITRE Corporation. CVE-2009-1235, 2009.

[Mohan et al. 2019] Mohan J, Martinez A, Ponnapalli S, Raju P, Chidambaram V. CrashMonkey and ACE: Systematically Testing File-System Crash Consistency. *ACM Transactions on Storage*, 15(2), Article 14, 2019.

[Nethercote and Seward 2007] Nethercote N, Seward J. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89-100, 2007.

[Nossum and Casanovas 2016] Nossum V, Casasnovas Q. Filesystem Fuzzing with American Fuzzy Lop. *Proceedings of Vault Linux Storage and Filesystems Conference*, 2016.

[Pailoor et al. 2018] Pailoor S, Aday A, Jana S. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. *Proceedings of the 27th USENIX Security Symposium*, 2018.

[Peng and Payer 2020] Peng H, Payer M. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. *Proceedings of the 29th USENIX Security Symposium*, USENIX Security 2020.

[Purdila et al. 2010] Purdila O, Grijincu LA, Tapus N. LKL: The Linux Kernel Library. *Proceedings of the 9th RoEduNet IEEE International Conference*, 2010.

[Rodeh et al. 2013] Rodeh O, Bacik J, Mason C. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS),* 9(3), Article 9, 2013.

[Schumilo et al. 2017] Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. *Proceedings of the 26th USENIX Security Symposium*, 2017,

[SGI 2022] SGI, OSDL and Bull. Linux Test Project. https://github.com/linux-test-project/ltp, 2022.

[Song et al. 2019] Song D, Hetzelt F, Das D, Spensky C, Na Y, Volckaert S, Vigna G, Kruegel C, Seifert JP, Franz M. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. 2019.

[syzbot 2022] syzbot, Google. https://Syzkaller.appspot.com/upstream, 2022.

[Syzkaller 2022] Syzkaller, Google. https://github.com/google/Syzkaller, 2022.

[Syzkaller 2022a] Syzkaller, Syscall Descriptions, https://github.com/google/Syzkaller/blob/master/docs/syscall_descriptions.md, 2022a.

[Wang et al. 2021] Wang D, Zhang Z, Zhang H, Qian Z, Krishnamurthy SV, Abu-Ghazaleh N. Beating Kernel Fuzzing Odds with Reinforcement Learning. *Proceedings of the 30th USENIX Security Symposium*, 2021.

[Wen et al. 2020] Wen C, Wang H, Li Y, Qin S, Liu Y, Xu Z, Chen H, Xie X, Pu G, Liu T. Memlock: Memory usage guided fuzzing. *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[Xu et al. 2019] Xu W, Moon H, Kashyap S, Tseng PN, Kim T. Fuzzing File Systems via Two-Dimensional Input Space Exploration. *Proceedings of the 2019* IEEE Symposium on Security and Privacy (SP), 2019.

[Xu et al. 2020] Xu M, Kashyap S, Zhao H, Kim T. Krace: Data race fuzzing for kernel file systems. *Proceedings of 2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[Zalewski 2018] Zalewski M. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl, 2018.

[Zhao et al. 2022]. Zhao B, Li Z, Qin S, Ma Z, Yuan M, Zhu, W, Zhang C. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. *Proceedings of the 31st USENIX Security Symposium*, 2022.