2019 IEEE Symposium on Security and Privacy

Fuzzing File Systems via Two-Dimensional Input Space Exploration

Wen Xu Hyungon Moon[†] Sanidhya Kashyap Po-Ning Tseng Taesoo Kim

Georgia Institute of Technology
[†]*Ulsan National Institute of Science and Technology*

***Abstract*—File systems, a basic building block of an OS, are too big and too complex to be bug free. Nevertheless, file systems rely on regular stress-testing tools and formal checkers to find bugs, which are limited due to the ever-increasing complexity of both file systems and OSes. Thus, fuzzing, proven to be an effective and a practical approach, becomes a preferable choice, as it does not need much knowledge about a target. However, three main challenges exist in fuzzing file systems: mutating a large image blob that degrades overall performance, generating image-dependent file operations, and reproducing found bugs, which is difficult for existing OS fuzzers.**

Hence, we present JANUS, the first feedback-driven fuzzer that explores the two-dimensional input space of a file system, i.e., mutating metadata on a large image, while emitting image-directed file operations. In addition, JANUS relies on a library OS rather than on traditional VMs for fuzzing, which enables JANUS to load a fresh copy of the OS, thereby leading to better reproducibility of bugs. We evaluate JANUS on eight file systems and found 90 bugs in the upstream Linux kernel, 62 of which have been acknowledged. Forty-three bugs have been fixed with 32 CVEs assigned. In addition, JANUS achieves higher code coverage on all the file systems after fuzzing 12 hours, when compared with the state-of-the-art fuzzer Syzkaller for fuzzing file systems. JANUS visits 4.19× and 2.01× more code paths in `Btrfs` and `ext4`, respectively. Moreover, JANUS is able to reproduce 88–100% of the crashes, while Syzkaller fails on all of them.

I. Introduction

File systems are one of the most basic system services of an operating system that play an important role in managing the files of users and tolerating system crashes without losing data consistency. Currently, most of the conventional file systems, such as `ext4` [8], `XFS` [64], `Btrfs` [59], and `F2FS` [30], run in the OS kernel. Hence, bugs in file systems cause devastating errors, such as system reboots, OS deadlock, and unrecoverable errors of the whole file system image. In addition, they also pose severe security threats. For instance, attackers exploit various file system issues by mounting a crafted disk image [28] or invoking vulnerable file system-specific operations [37] to achieve code execution or privilege escalation on victim machines. However, manually eliminating every bug in a file system that has sheer complexity is a challenge, even for an expert. For example, the latest implementation of `ext4` in Linux v4.18 comprises 50K lines of code, while that of `Btrfs` is nearly 130K LoC. At the same time, many widely used file systems are still under active development. File system developers consistently optimize performance [72] and add new features [11, 29], meanwhile introducing new bugs [26, 27, 40].

To automatically discover these potential bugs, most file systems in development rely on the known stress-testing frameworks (xfstests [63], `fsck` [56, 69], Linux Test Project [62], etc.) that mostly focus on the regression of file systems with minimal integrity checks. For example, one of the bugs we found in `ext4` (i.e., CVE-2018-10880) crashes the kernel by moving a critical extended attribute out of the `inode` structure. We trigger this bug by mounting a normal `ext4` image formatted with `inline_data`, which bypasses integrity checks in both xfstests and `fsck`. In addition, some prior works have applied model checking [73, 74] to find file system bugs, which requires a deep understanding of both the file system and OS states. This is now impractical due to the increasing complexity of modern OSes [2, 35, 68]. On the other hand, most of the verified file systems [6, 9] are too immature to adopt in practice.

Another approach—fuzzing—is gaining traction. Fuzzing not only requires minimal knowledge about the target software, but also is an *effective* and a *practical* approach that has found thousands of vulnerabilities [15, 18, 34, 76]. Hence, fuzzing is a viable approach to automatically discover bugs in a wide range of file systems (e.g., 54 in the Linux kernel). However, unlike other ordinary targets, fuzzing file systems is dependent on two inputs: a mounted disk image and a sequence of file operations (i.e., system calls) that are executed on the mounted image. Existing fuzzers either focus on mutating images as ordinary binary inputs [21, 48, 57, 61] or generating random sets of file operation-specific system calls [20, 25, 46]. Unfortunately, they all fail to efficiently and comprehensively test file systems because of the following three challenges.

First, a disk image is a large binary blob that is structured but complicated, and the *minimum* size can be almost 100× larger than the *maximum* preferred size of general fuzzers [76], which dramatically degrades the fuzzing throughput [21, 57, 61] due to the heavy I/O involved in mutating images. Another issue related to blob mutation is that existing fuzzers [20, 48] mutate only non-zero chunks in an image. This approach is unsound because these fuzzers do not exploit the properties of structured data, i.e., file system layout, in which mutating metadata blocks is more effective than mutating data blocks. In addition, without any knowledge about the file system layout, existing fuzzers also fail to fix any metadata checksum after corrupting metadata blocks. The second challenge is that file operations are context-aware workloads, i.e., a dependence exists between an image and the file operations executed on

DOI 10.1109/SP.2019.00035

it. In particular, the real-time status of a mounted file system determines which file objects a set of system calls can operate on, and the invocation of a particular system call brings changes to the object being operated on. Unfortunately, existing system call fuzzers [20, 25, 46], which independently generate random system calls with hard-coded file paths, fail to emit meaningful sequences of file operations, and cover deep code paths of a file system. The third issue with existing fuzzers is the aspect of reproducing found bugs. Most of the existing fuzzers that target OSes [20, 25, 46] or file systems [48] test generated inputs without reloading a fresh copy of the OS instance or file system image, i.e., they do not use *a non-aging OS and file system*. The reason they do not re-initialize the OS or file systems is that they rely on VM, QEMU, or user-mode Linux (UML) [13] instances that take seconds to reload a fresh copy. To overcome this issue, they reuse these instances, thereby leading to dirty OS states, which eventually results in unstable executions and irreproducible bugs.

We address the aforementioned challenges with JANUS, an evolutionary feedback-driven fuzzer, that effectively explores the two-dimensional input space of a disk file system. JANUS addresses the first problem by exploiting the structured data property in the form of metadata, i.e., it mutates only metadata blocks of a seed image, thereby drastically pruning the searching space of the input. Second, we propose the *image-directed syscall fuzzing* technique to fuzz file operations, i.e., JANUS not only stores generated system calls but also deduces the runtime status of every file object on the image after these system calls complete. JANUS then uses the speculated status as feedback to generate a new set of system calls, thereby emitting context-aware workloads. During each fuzzing iteration, JANUS performs image fuzzing with higher priority and then invokes image-directed syscall fuzzing to fully explore a target file system. Finally, JANUS solves the reproducibility problem, which is tightly coupled with the scalability of OS fuzzing as well as OS aging, by always loading a fresh copy of the OS to test the file system-related OS functionalities with the help of a library OS (i.e., Linux Kernel Library [54]), running in user space.

With JANUS, we fuzzed eight popular file systems in the upstream Linux kernel (v4.16–v4.18) for four months. Our evaluation shows that JANUS achieves at most 4.19× more code coverage than the state-of-the-art OS fuzzer Syzkaller. Moreover, our choice of using a library OS enables us to reproduce 88-100% of crashes, while Syzkaller fails to reproduce any. Until now, we have successfully found 90 bugs, and developers have already acknowledged 62 of them, 43 of which have been fixed with 32 CVEs assigned.

This paper makes the following contributions:

- **Issues.** We identify three prominent issues of existing file systems fuzzers: (1) fuzzing a large blob image is inefficient; (2) fuzzers do not exploit the dependence between a file system image and file operations; (3) fuzzers use aging OSes and file systems, which results in irreproducible bugs.
- **Approach.** We design and implement an evolutionary file system fuzzer, called JANUS, that efficiently mutates metadata blocks in a large seed image while generating image-directed workloads to extensively explore a target file system. JANUS further leverages a library OS (i.e., LKL) other than a VM to test OS functionalities, so as to provide a clean-slate OS image in a matter of milliseconds.
- **Impact.** We evaluate JANUS on eight file systems and find 90 bugs in the upstream kernel, 62 and 43 of which have been acknowledged and patched with 32 CVEs assigned. Moreover, JANUS outperforms Syzkaller regarding code coverage on all selected file systems. In particular, JANUS eventually visits 4.19× and 2.01× more code paths than Syzkaller when fuzzing `Btrfs` and `ext4`, respectively, for 12 hours. Meanwhile, JANUS can reproduce 88-100% of the found crashes, while Syzkaller fails to reproduce any.

Threat Model. In this work, we assume that an attacker is privileged to mount a fully crafted disk image on a target machine and operate files stored on the image to exploit security bugs in an in-kernel file system. Practical ways exist for an attacker to achieve this without root privilege, including: (1) Auto-mounting. Modern OSes automatically mount an untrusted plugged-in drive if it supports the corresponding file system, which is exploited by several infamous attacks such as Stuxnet [28], "evil maid attack" [60], etc.; (2) Unprivileged mounts. macOS allows a non-root user to mount an Apple disk image applying various file systems such as HFS, HFS+, APFS, etc., and a number of bugs are found in these file systems that lead to memory read restriction bypass and code execution in the kernel [38, 39, 77]. Linux also allows unprivileged users to mount any file system with `FS_USERNS_MOUNT` in a user namespace [12].

II. BACKGROUND AND MOTIVATION

Commodity OSes usually implement a disk file system as a kernel module. Users are tasked with mounting the *large-size* and *formatted* image and manage data via *file operations*. In this section, we first describe general fuzzing approaches (§II-A) and existing file system fuzzers (§II-B). Later, we explain why they all fail to efficiently test file systems. We then summarize the challenges and potential opportunities in file system fuzzing (§II-C).

A. A Primer on Fuzzing

Fuzzing is a popular software-testing method that repeatedly generates new inputs and injects them into a target program to trigger bugs. It is one of the most effective approaches in practice to find security bugs in modern software. For example, the state-of-the-art fuzzer AFL [76] and its variants [4, 5, 15, 51], have discovered numerous bugs in open-source software. To effectively explore a target program, recent fuzzers leverage the past code coverage to later direct the input generation. Moreover, software such as an OS is the most critical program, as discovered bugs allow privilege escalation on a target machine. To fuzz OSes, several frameworks [20, 46, 61] extend

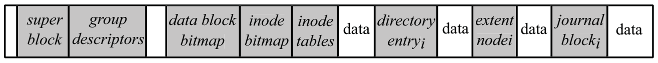

Fig. 1: The on-disk layout of an `ext4` image. The gray blocks shows metadata in use, which occupies merely 1% of the image size. Some of them, including extent tree nodes, directory entries, and journal blocks, are scattered in the image, while others (e.g., superblock, group descriptors, etc.) are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

File system	ext4	XFS	Btrfs	F2FS	GFS2	ReiserFS	NTFS	AFL
Min. size (MB)	2	16	100	38	16	33	1	1

TABLE I: The minimal size of a block device allowed to be formatted by various file systems with default options along with enabled journaling or logging. Most of the image sizes exceed the size of a fuzzing input suggested by AFL (1MB).

feedback-driven fuzzing approaches to trigger kernel bugs by invoking randomly generated system calls.

B. File System Fuzzing

A disk file system has two-dimensional input space: (1) the structured *file system image format*; and (2) *file operations* that users invoke to access files stored on a mounted image. Several file system fuzzing tools use a generic fuzzing infrastructure to target either images or file operations.

1) Disk Image Fuzzer: A disk image is a *large structured* binary blob. The blob has (1) user data and (2) several management structures, called *metadata*, that a file system needs to access, load, recover, and search data or to fulfill other specific requirements of a file system. Figure 1 presents the on-disk layout of a typical `ext4` image.[1] However, the size of metadata constitutes merely 1% of the image size [1]. On the other hand, the minimal size of a valid image can be 100 MBs (see Table I), which further increases on enabling certain features. Three issues occur with an image as a fuzzing input: (1) Large input size leads to an exponential increase in the input space exploration. Meanwhile, important metadata are mutated infrequently. (2) A fuzzer performs frequent read and write operations on input files. When fuzzing a disk image, it repeatedly reads during mutation, writes after mutation, and saves the image if necessary. As a result, the large size of a disk image slows down essential file operations, leading to huge performance overhead. (3) Finally, to detect metadata corruption, several file systems (e.g., `XFS` v5, `GFS2`, `F2FS`, etc.) introduce checksums to protect on-disk metadata. Hence, the kernel rejects a corrupt image, with mutated metadata blocks without correct checksums, during initialization.

Disk image fuzzers [21, 48, 57, 61, 69] enforce a file system to mount and execute a sequence of file operations on the mutated disk images to trigger file system-specific bugs. Early fuzzers [21, 57] ineffectively mutate bytes at random offsets in a valid image to generate new images or mutate bytes in metadata blocks only. These approaches incur heavy disk I/O from loading and saving entire images. Moreover, these blind fuzzing techniques generate poor-quality images without utilizing past coverage. To overcome this, most recent fuzzers [20, 48] are driven by code coverage. Moreover, they extract all the non-zero chunks in a seed image for mutation. This approach touches most of the metadata blocks and improves fuzzing performance by decreasing input size. Nevertheless, these non-zero chunks not only contain non-zero data blocks but also discard the zero initialized metadata blocks, which results in sub-optimal file system fuzzing. In addition, as metadata blocks are not precisely located, this approach fails to fix their checksums.

2) File Operation Fuzzer: Since file systems are part of the OS, a general approach to fuzz them is to invoke a set of system calls [20, 25, 46]. Although porting these fuzzers to target file system operations is straightforward, they fail to efficiently fuzz file systems for two reasons: First, file operations modify only file objects (e.g., directories, symbolic links, etc.) that exist on the image, and a completed operation affects particular objects. However, existing OS fuzzers do not consider the dynamic dependence between an image and file operations, as they *blindly* generate system calls, that explore a file system superficially. For example, the state-of-the-art OS fuzzer, Syzkaller, generates system calls based upon static grammar rules describing the data types of every argument and return value for every target system call. Therefore, Syzkaller is able to generate a single semantically correct system call but fails to explore the collective behavior of a set of system calls and the modified file system image. For instance, Syzkaller may emit multiples of `open()` calls on a file with its old path that has been either renamed (`rename()`) or removed (`unlink()`).

Second, existing OS fuzzers [20, 61] mostly use virtualized instances (e.g., KVM, QEMU, etc.) to run a target OS without reloading a fresh copy of the OS or file system for every testing input for the sake of performance. Unfortunately, fuzzing with an aging OS or file systems has two issues: (1) The execution of an aging OS becomes non-deterministic after processing numerous system calls. For example, `kmalloc()` which depends on prior allocations, behaves differently across runs. Sometimes a kernel component (e.g., journaling system) quietly fails and detaches from the OS without triggering any file system crash during a long-run fuzzing. (2) A bug found by these fuzzers accumulates the impact of thousands of invoked system calls, which impedes the generation of a stable proof-of-concept for developers to reproduce the bug and debug it [19].

3) File System Fuzzer: As mentioned before, most fuzzers either fuzz a binary input [34, 76] or use a sequence of system calls to fuzz the OS [20, 25]. However, to fuzz a file system, we need to mutate two inputs: (1) the binary image (i.e., a file system image) and (2) the corresponding workload (i.e., a set of file system specific system calls). Unfortunately, combining these two existing fuzzing techniques is not straightforward. Recently, Syzkaller tried to achieve both by mutating non-zero chunks in an image, while independently generating *context-unaware* workloads to test the mutated image, which is still unsound and ineffective.

C. Challenges of Fuzzing a File System

We summarize a set of challenges of fuzzing file systems in the Linux kernel, and present our insights in designing

[1]We have listed its file hierarchy in Figure 9 including files and directories.

JANUS to overcome these challenges. Note that our insights are applicable to other OSes as well.

Handling large disk images as input. An image fuzzer should effectively fuzz a complicated, large disk image by (1) mutating scattered metadata in the image with checksum, and (2) mitigating frequent disk I/O due to input manipulation. Unfortunately, current fuzzers fail to address these issues simultaneously (see §II-B1). An ideal image fuzzer should target only the metadata, rather than the entire disk image, and must fix the checksum for any mutated metadata structure.

Missing context-aware workloads. File system-aware workloads directly affect the image. In particular, valid file operations modify file objects on an image (e.g., `open()` creates a new file and `unlink()` removes one link of an existing file) at runtime. However, existing fuzzers rely on the predefined image information (i.e., valid file and directory paths on a seed image) to generate system calls, and thereby fail to comprehensively test all the accessible file objects in a target file system at runtime (§II-B2). Therefore, a better approach is to maintain the runtime status of every file object on an image after performing past file operations for generating new ones.

Exploring input space in two dimensions. A file system processes two types of inputs, including disk images and file operations which are organized in completely different formats (i.e., binary blob versus sequential operations), but have an implicit connection between them. To fully explore a file system, a fuzzer should mutate both of them, which is not supported by existing fuzzers. Thus, we aim to propose a hybrid approach that explores both dimensions by fuzzing image bytes and file operations simultaneously.

Reproducing found crashes. Traditional OS fuzzers use virtualized instances to test OS functionalities. However, to avoid the expensive cost of rebooting a VM or reverting a snapshot, they re-uses an OS or file system instance across multiple runs, which leads to *unstable kernel executions* and *irreproducible bugs* (see §II-B2). This issue can be overcome by leveraging a library OS [53, 54] that provides the exact OS behavior and re-initializes the OS states within milliseconds.

III. DESIGN

A. Overview

JANUS is a feedback-driven fuzzer that mutates the metadata of a seed image, while generating context-aware file operations (i.e., system calls) to comprehensively explore a file system. In general, JANUS adopts the following design choices to resolve the aforementioned challenges regarding file system fuzzing (see §II-C). First, JANUS merely stores the metadata extracted from the seed image as its mutation target, which is critical for a file system to manage user data. In addition, JANUS re-calculates every metadata checksum value after mutation. Since the metadata occupy a small space (1%), the size of an input test case is much smaller than that of an entire disk image, which enables high fuzzing throughput. Second, JANUS does not rely on manually specified information about the files stored on a seed image, as it becomes stale over time

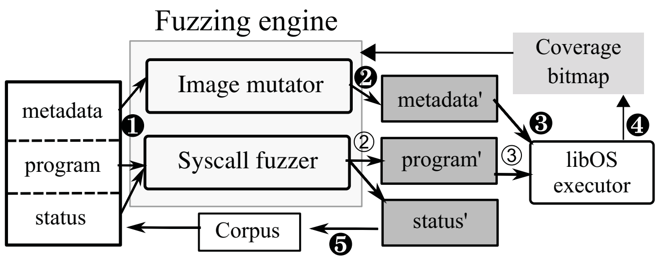

Fig. 2: An overview of JANUS. In each fuzzing iteration, JANUS loads a test case from its working corpus, which consists of three parts: the metadata of a seed image, a program containing a list of file operations, and the speculated image status at runtime after executing the program (❶). Then the fuzzing engine of JANUS mutates the test case in two directions: (1) The image mutator randomly mutates the metadata, and the fuzzing engine outputs the mutated metadata along with the intact program for testing (❷), or (2) The syscall fuzzer mutates existing system calls in the program or appends new ones, and updates the image status correspondingly as the workload changes (②). In this case, the fuzzing engine outputs the intact metadata and the newly generated program. Next, JANUS releases the output metadata into a full-size image (❸) and delivers the image with the output program (③) to a library OS based executor. The executor mounts the image and executes the program, whose execution trace is profiled into a bitmap shared with the fuzzing engine of JANUS (❹). If new code paths are discovered, the output metadata and program, and the updated image status are packed as a new test case and saved into the corpus for future mutation (❺).

and results in ineffective test case generation. Instead, JANUS generates new file operations based upon the deduced status of an image after completing old ones in a workload. Third, JANUS manages to explore the two-dimensional input space of a file system by wisely scheduling image fuzzing and file operation fuzzing. Considering the fact that the original image determines the initial state of a file system and affects the executions of the foremost file operations, JANUS makes the first effort to mutate image bytes. Lastly, JANUS relies on a library OS to test kernel functions in user space. A library OS instance runs as a user application, which can be re-launched with negligible overhead, and thereby helps to increase the chance of reproducing a found bug.

Figure 2 presents the detailed design of JANUS. A binary input for JANUS consists of three parts: (1) a binary blob comprising the metadata blocks of a seed image, (2) a serialized program (i.e., file system workload) that describes a sequence of system calls, and (3) the speculated image status after the program operates the image. In the beginning, JANUS relies on a file system-specific parser to extract metadata from a seed image. JANUS also inspects the seed image to retrieve initial image status and generate starting programs. The original metadata, along with the image status and the program, are packed as a test case and are saved into JANUS's working corpus. JANUS initiates fuzzing with both the image mutator and the system call fuzzer by selecting a test case from the corpus (❶) for exploring the two-dimensional input space in an infinite loop. First, the fuzzing engine invokes the image mutator to flip the bytes of the metadata blob in several ways

```
# Class Janus
def generate_corpus(self, image, fstype):
    self.image_buffer = read_image(image)
    meta_blocks = self.img_parser.parse_image(image, fstype)
    meta_buffer = ""
    for meta_block in meta_blocks:
        self.meta_blocks[i].offset = meta_block.offset
        self.meta_blocks[i].size = meta_block.size
        if meta_block.has_csum:
            self.meta_blocks[i].csum_offset = meta_block.csum_offset
        else:
            self.meta_blocks[i].csum_offset = None
        meta_buffer += meta_block.buffer

    file_objs = self.inspect_image(image)
    program = Program()
    status = Status(file_objs)
    self.sys_fuzzer.initialize(program, status)
    for file_obj in file_objs:
        (new_program, new_status) = \
            self.sys_fuzzer.generate_syscall(SYS_OPEN, [file_obj])
        self.add_into_corpus((meta_buffer, new_program, new_status))
```

Fig. 3: Pseudo-code of how JANUS generates the initial corpus given a seed image.

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

Fig. 4: Pseudo-code of how JANUS randomly mutates metadata blocks.

and outputs mutated blobs (❷). At the same time, the program in the test case remains unchanged. Later on, the system call fuzzer enables JANUS to either mutate the argument values of existing system calls in the program or append new ones to the program. The system call fuzzer also produces new image status according to the newly generated program (②). Meanwhile, the metadata part remains intact. The output metadata is combined with other unchanged parts (i.e., user data) to produce a full-size image with all the checksum values re-calculated by JANUS (❸). And the output program is also serialized and saved onto the disk (③). A user-space system call executor, which relies on a library OS, launches a new instance to mount the full-size image and perform the file operations involved in the program loaded from the disk (❹). The runtime path coverage of the executor is profiled into a bitmap shared with JANUS's fuzzing engine. The fuzzing engine inspects the bitmap; on discovering a new path, JANUS saves the shrunken image, the serialized program, and the speculated image status into one binary input for further mutation in successive runs (❺). Note that for each test case, JANUS always launches the image mutator first for certain rounds and invokes the system call fuzzer if no interesting test case is discovered.

We first describe how JANUS generates the starting test cases by parsing a seed image in §III-B. We then present how it fuzzes image bytes and generates file operations in §III-C and §III-D, respectively. More important, we describe how JANUS integrates two core fuzzers in §III-E. Finally, we present our new library OS-based environment for file system fuzzing in §III-F.

B. Building Corpus

JANUS relies on its image parser and system call fuzzer to build its initial corpus upon a seed image (see Figure 3). The first part of the test cases in the corpus is the essential metadata blocks of the seed image, which constitutes around 1% of the total size, thereby overcoming the challenges of fuzzing a disk image, as described in §II-C. Specifically, JANUS first maps the entire image into a memory buffer. Then a file system-specific image parser scans the image and locates all the on-disk metadata according to the specification of the applied file system. JANUS re-assembles these metadata into a shrunken blob for mutation afterward and records their sizes and in-image offsets. For any metadata structure protected by checksum, JANUS records the in-metadata offset of the checksum field recognized by the image parser. Second, the starting test cases also include the information of every file and directory on the image that allows JANUS to use that knowledge for generating context-aware workloads afterward. In particular, the system call fuzzer probes the seed image and retrieves the path, type (e.g., normal file, directory, symbolic link, FIFO file, etc.), and extended attributes of every file object on it, which are packed into every initial test case. Moreover, every initial test case involves a starting program that has a distinct system call generated by the system call fuzzer for mutation. To enlarge the overall coverage of the corpus, each randomly generated system call operates a unique file object (see §III-D for the details of program format and system call generation). The metadata and the file status of the seed image, along with a starting program together form an input test case, which is packed by JANUS and saved into the corpus on the disk for future fuzzing.

C. Fuzzing Images

JANUS relies on the image mutator to fuzz images. In particular, the image mutator loads the metadata blocks of a test case, and applies several common fuzzing strategies [76] (e.g., bit flipping, arithmetic operation on random bytes, etc.) to randomly mutate the bytes of the metadata, as described in Figure 4 (❷). Similar to existing fuzzers [75], JANUS prefers a group of specific integers (i.e., *interesting values* in Figure 4), such as `-1`, `0`, `INT_MAX`, etc., instead of purely random values to mutate the metadata. In our evaluation, these special values enable the image mutator to produce more corner cases, which are not correctly handled by the file system (e.g., bug #1, #6, #14, #28, #33, etc. in Table VI found by JANUS) and also more extreme cases that increase the probability of crashing

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

Fig. 5: Pseudo-code of how JANUS releases the mutated metadata blocks back to a full-size image for testing.

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

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

Fig. 6: Pseudo-code of how JANUS randomly mutates existing system calls and generate new ones given a program.

the kernel by triggering a specific bug at runtime (e.g., most of the out-of-bound access bugs discovered by JANUS).

After mutating the entire metadata blob, JANUS copies each metadata block in the blob back to its corresponding position inside the memory buffer, which stores the original full-size image (❸). To maintain the sanctity of the image, the image parser recalculates the checksum value of every metadata block by following the specific algorithm adopted by the target file system, and fills the value at the recorded offset of the checksum field.

D. Fuzzing File Operations

The system call fuzzer enables JANUS to generate image-directed workloads to effectively explore how a file system handles various file operations requested by users. First, we present the structure of a program manipulated by the system call fuzzer. A program includes a list of ordered system calls that modifies the mutated image and maintains a variable bank that stores the variables used by system calls. JANUS describes each system call as a tuple of the syscall number, argument values, and a return value. If any argument value or return value is not a simple constant but a variable, JANUS presents it as an index pointing to the variable stored in the variable bank. In addition, the program also includes a list of active file descriptors that are opened and have not been closed by the program.

Similar to existing fuzzers (e.g., Syzkaller), the system call fuzzer generates new programs from an input program in two ways: (1) Syscall mutation. The system call fuzzer randomly selects one system call in the program, and generates a list of new values to replace the old value of a randomly selected argument; (2) Syscall generation. The system call fuzzer appends a new system call to the program, whose arguments have randomly generated values. In particular, JANUS adopts the same strategies that Syzkaller uses to generate values for the trivial arguments of a system call. The candidate values of these arguments are independent of our speculated runtime status. For any argument that has a clearly defined set of its available values, JANUS randomly selects values from the set for it. (e.g., `int whence` for `lseek()`). Moreover, JANUS generates random numbers in a certain range for the arguments of an integer type (e.g., `size_t count` for `write()`). Furthermore, a number of file operations requires an argument of a pointer type. Such a pointer normally points to a buffer that is used to store either user data (e.g., `void *buf` for `write()`) or kernel output (e.g., `void *buf` for `read()`). For the former case, the system call fuzzer declares an array filled with random values for the argument. A fixed array is always used in the latter case, since JANUS is not driven by what the kernel outputs at runtime except for its code coverage.

Nevertheless, for those non-trivial arguments whose proper values depend on the running context of a file system, JANUS generates their values based not only on their expected types, but more important, on our maintained status by following mainly three rules: (1) If a file descriptor is required, the system call fuzzer randomly picks an opened file descriptor of proper type. For instance, `write()` requires a normal file descriptor, while `getdents()` asks for the file descriptor of a directory; (2) If a path is required, the system call fuzzer randomly selects the path of an existing file or directory, or a stale file or directory that is removed by recent operations. For instance, JANUS provides the path of a normal file or a directory to `rename()`, but delivers only that of a valid directory required by `rmdir()`. If the path is used to create a new file or directory, JANUS may also randomly generate a brand new path that is located under an existing directory; (3) If a system call operates the existing extended attribute of a particular file (e.g., `getxattr()` and `setxattr()`), the system call fuzzer randomly picks a recorded extended attribute name of the file. The generation strategies enable JANUS to emit context-aware workloads on fresh file objects that are free of runtime errors and achieve high code coverage.

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

```python
# Class Janus
def run_one(self, buffer, program, status):
    cov, lkl_status = self.lkl_test(self.image_buffer, program)
    if lkl_status == CRASH:
        self.save_crash((buffer, program))
    elif self.has_new_path(cov):
        self.add_into_corpus((buffer, program, status))
        return True
    return False

def fuzz_one(self):
    (meta_buffer, program, status) = ctx.pick_from_corpus()
    found_new = False
    for _ in range(IMAGE_MUTATE_CYCLES):
        mutated_buffer = self.img_mutator.mutate_image(meta_buffer)
        if self.run_one(mutated_buffer, program, status):
            found_new = True
    if found_new: return
    self.release_image(ctx, meta_buffer)
    self.sys_fuzzer.initialize(program, status)
    for _ in range(SYSCALL_MUTATE_CYCLES):
        new_program = self.sys_fuzzer.mutate_syscall()
        if self.run_one(meta_buffer, new_program, status):
            found_new = True
    if found_new: return
    for _ in range(SYSCALL_GENERATE_CYCLES):
        (new_program, new_status) = \
            self.sys_fuzzer.generate_syscall()
        self.run_one(meta_buffer, new_program, new_status)
```

Fig. 7: Pseudo-code of one fuzzing iteration in JANUS.

Note that in the current design, JANUS maintains only the speculated image status after completing the execution of a program. Therefore, JANUS avoids any mutation on the existing arguments that result in potential changes to the image status. For instance, JANUS may mutate `fd` of a `write()` in the program while never touching `path` of `unlink()`, since such a mutation may invalidate the system calls after the mutated ones (e.g., changing `unlink("A")` to `unlink("B")` affect all the existing file operations afterward on file `B` in a test case).

E. Exploring Two-Dimensional Input Space

To fuzz both metadata and system calls together, JANUS schedules its two core fuzzers in order. Figure 7 describes one fuzzing iteration of JANUS. Specifically, for an input test case, which contains a shrunken image and a program, JANUS first launches the image mutator to mutate random bytes on the shrunken image. If no new code path is discovered with the unchanged program, JANUS invokes the system call fuzzer to mutate the argument values of an existing system call in the program for certain rounds. If still no new code path is explored, JANUS eventually tries to append new system calls to the program. Note that rounds in every fuzzing stage are user defined.

Scheduling image fuzzing and file operation fuzzing in such an order is effective as follows: (1) The extracted metadata indicate the initial state of an image, whose impacts on the executions of file operations gradually decreases when the image has been operated by several system calls. Hence, JANUS always tries to mutate metadata first. (2) Introducing new file operations exponentially increases the mutation space of a program and may also erase the changes from past operations of the image. Therefore, JANUS prefers mutating existing system calls rather than generating new ones.

Component	**LoC**	**Languange**
Fuzzing engine		
Image parser (8 file systems)	5,229	C++
Image inspector	141	Python
Program serializer	1,163	C++
Syscall fuzzer	3,137	C++
Other AFL changes	497	C
LKL changes		
Shared image buffer	16	C
KASAN	804	C
Instrumentation tool	360	C++
LKL-based executor	851	C++
PoC generator	1,108	C++, Python

TABLE II: Implementation complexity of JANUS, including the changes to AFL and LKL for file system fuzzing. Since we directly reuse the existing binary mutation algorithms in AFL for the image mutator, we omit its code size.

F. Library OS based Kernel Execution Environment

To avoid using an aging OS or file system that results in unstable executions and irreproducible bugs (see §II-C), JANUS relies on a library OS based application (i.e., executor) to fuzz OS functionalities. Specifically, JANUS forks a new instance of the executor to test every newly generated image and workload from the fuzzing engine (❹). Note that forking a user application incurs negligible time compared with resetting a VM instance. Hence, JANUS guarantees a clean-slate kernel for every test case with low overhead. Moreover, as both fuzzing engine and executor run in user space on one machine, sharing input files and coverage bitmap between them is straightforward, which is challenging for VM-based fuzzers that run the fuzzing engine outside VM instances. In addition, a library OS instance requires far less computing resources compared with any type of VMs. Therefore, we can deploy JANUS instances on a large scale without severe contention.

IV. IMPLEMENTATION

We implement JANUS as a variant of AFL (version 2.52b). JANUS adopts the basic infrastructure of AFL, including the forkserver, coverage bitmap, and test-case scheduling algorithm. We extend AFL with the the image mutator and the system call fuzzer. In addition, we implement an image inspector to build the initial corpus from a seed image and a program serializer for delivering generated programs between memory and working corpus. Furthermore, we implement an executor based on Linux Kernel Library (LKL) to test newly generated images and workloads. Note that we also modify LKL to support the kernel address sanitizer (KASAN) [17], which is widely adopted by OS fuzzers to detect memory errors. For ease of reproducing bugs in a real environment, we also implement a Proof-of-Concept (PoC) generator that produces a full-size image along with a compilable C program from a serialized test case. Table II presents the lines of code (LoC) of each components of JANUS. In this section, we describe the implementation details of several main components.

Image parser and image mutator. We implement the image parser as a dynamic library to locate metadata and identify checksums on a seed image. Currently, the image parser supports parsing the disk images of eight widely used file systems on Linux, including `ext4`, `XFS`, `Btrfs`, `F2FS`, `GFS2`, `HFS+`, `ReiserFS`, and `VFAT`. Our implementation of the image parser refers to the user-space utilities (e.g., `mkfs` and `fsck`) of those file systems. We also implement the image mutator, which randomly mutates the bytes of a shrunken image through eight strategies (see Figure 4). We directly port the implementation of these mutation strategies from AFL in JANUS.

Image inspector. We implement an image inspector for JANUS, which iterates files and directories on a seed image, and records their in-image paths, types, and extended attributes for building initial test cases (see §III-B).

Program serializer. We describe newly generated programs and updated status in a serializable format (see Figure 12) and implement a corresponding program serializer. The serializer loads them from the disk into the memory for fuzzing and testing, and saves them from memory onto the disk for bookkeeping.

System call fuzzer. The system call fuzzer is implemented as a new extension for AFL, which is invoked by JANUS when image mutation fails to make progress. The system call fuzzer receives a deserialized program and the corresponding status, and outputs new programs and updated status through system call mutation or system call generation (see §III-D). Currently, JANUS supports generating and mutating 34 system calls designed for fundamental file operations (see §F). A number of system calls related to file operations to a certain extent but mainly realized at the VFS (virtual file system) layer (e.g., `dup()`, `splice()`, `tee()`, etc.) are not worth being tested and are excluded by JANUS.

In our implementation, JANUS basically mutates metadata in a test case for 256 rounds, which is the default setting for the havoc stage (i.e., nondeterministic mutation) in AFL. If the code coverage fails to increase, JANUS tries to mutate existing system calls for 128 rounds and appends new ones for another 64 rounds. JANUS spends more effort on image mutation due to its higher priority when exploring the two-dimensional input space (see §III-E).

LKL-based executor. We build our executor for JANUS upon Linux Kernel Library (LKL), which is a typical library OS that exposes kernel interfaces to user-space programs. Figure 11 presents a code example of using LKL system calls to operate an `ext4` image. The official LKL currently works with Linux kernel v4.16 and we port it to be compatible with recent versions, including v4.17 and v4.18. To achieve AFL-style path coverage at runtime, we implement a `GCC` wrapper to selectively instrument the source files of a target file system when building LKL. Furthermore, we implement a user application (i.e., the executor) linked with LKL as the fuzzing target of JANUS. For a generated test case, the executor forks a new instance through the forkserver and invokes LKL system calls to mount an image mutated by the image mutator and perform a sequence of file operations generated by the system call fuzzer.

As flushing a full-size image onto the disk every time takes much time, we introduce a persistent memory buffer shared between JANUS's fuzzing engine and the LKL-based executor to store the image (i.e., `ctx.image_buffer` in Figure 3). The LKL's block device driver underlying a file system is then modified to access the memory buffer instead of the image file on the disk when acquiring any image data. Moreover, we apply the Copy-on-Write (CoW) technique at runtime to guarantee that besides the mutated blocks, other parts inside the image buffer never change when the image buffer is operated by the generated workload. Specifically, when the device driver tends to flush any byte back to a block on the image at runtime, the block is duplicated for modification and later accesses from LKL. In addition, we port the kernel address sanitizer (KASAN) to LKL, which can effectively detect memory errors at runtime. KASAN allocates shadow memory at runtime to record whether each byte of the original memory is safe to access. Note that KASAN relies on MMU to translate an address to its corresponding shadow address, which is not supported by LKL. Hence, we reserve the shadow memory space and build the mappings from the memory space of LKL to the shadow memory at LKL's boot time.

V. EVALUATION

In this section, we evaluate the effectiveness of JANUS in terms of its ability to find bugs in the latest file systems and achieve higher code coverage than existing file system fuzzers. In particular, we answer the following questions:

- **Q1:** How effective is JANUS in discovering previously unknown bugs in file systems? (§V-A)
- **Q2:** How effective is JANUS in exploring (1) the state of file system images, (2) file operations, and (3) the two-dimensional input space including images and file operations? (§V-B, §V-C, §V-D)
- **Q3:** Is the library OS based executor more effective in reproducing crashes than traditional VMs? (§V-E)
- **Q4:** Besides finding new bugs, what else can JANUS contribute to the file system community? (§V-F)

Experimental Setup. We evaluate JANUS on a 2-socket, 24-core machine running Ubuntu 16.04 with Intel Xeon E5-2670 processors and 256GB memory. We use JANUS to fuzz file systems in Linux v4.18-rc1, unless otherwise stated. In particular, we test eight file systems including `ext4`, `XFS`, `Btrfs`, `F2FS`, `GFS2`, `HFS+`, `ReiserFS`, and `VFAT`. We create a seed image for each file system that has the on-disk file organization shown in Figure 9 with most features enabled except `ext4` and `XFS`. For `ext4`, we create two seed images: one compatible with `ext2/3` and the other with `ext4` features. Similarly, we do the same for `XFS` representing `XFS` v4 and `XFS` v5, which introduces on-disk checksums to enforce metadata integrity. In total, we evaluate 10 seed images. In addition, we compare our results with Syzkaller (commit ID 9be5aa1), which is the state-of-the-

File Systems	#Reported	#Confirmed	#Fixed	#Patches	#CVEs
ext4	18	16	16	20	13
XFS	17	11	7	9	5
Btrfs	9	9	8	10	5
F2FS	11	11	11	12	8
GFS2	14	0	0	0	0
HFS+	8	7	1	1	1
ReiserFS	13	8	0	0	0
VFAT	0	0	0	0	0
Total	90	62	43	52	32

TABLE III: An overview of bugs found by JANUS in eight widely-used file systems in upstream Linux kernels. The column **#Reported** shows the number of bugs reported to the Linux kernel community; **#Confirmed** presents the number of reported bugs that are previously unknown and confirmed by kernel developers; **#Fixed** indicates the number of bugs that have already been fixed, at least in the development branch, and **#Patches** reports the number of git commits for fixing found bugs; **#CVEs** lists the number of CVEs assigned for confirmed bugs.

art OS fuzzer. We run Syzkaller with KVM instances, each of which has two cores and 2GB of memory.

Note that Syzkaller relies on KCOV to profile code coverage, while JANUS relies on the method of AFL. For an apples-to-apples comparison between Syzkaller and JANUS, after fuzzing 12 hours, we mount every image mutated by JANUS, and execute the corresponding program generated by JANUS on a KCOV-enabled kernel to get the KCOV-style coverage. (see Appendix §B for the details of AFL- and KCOV- style code coverage).

A. Bug Discovery in the Upstream File Systems

We intermittently ran JANUS for four months (i.e., from April 2018 to July 2018) to fuzz the aforementioned file systems in upstream kernels from v4.16 to v4.18. Over the span of few days to a week, we ran three instances of JANUS to test each file system. JANUS found 90 unique bugs that resulted in kernel panics or deadlocks, which we reported to the Linux kernel community. We differentiated bugs on the basis of KASAN reports and call stack traces. Among them, developers confirmed 62 as previously unknown bugs, including 36 in `ext4`, `XFS`, and `Btrfs`—the three most widely used file systems on Linux. So far, developers have already fixed 43 bugs with 52 distinct patches, and also assigned 32 CVEs (see Table III). Another important finding is that some bugs, (e.g., four bugs related to log recovery in `XFS` and six bugs about extended attributes in `HFS+`) are not going to be fixed by developers in the near future, as these bugs require large-scale code refactoring. In addition, `ReiserFS` developers will not fix five bugs that lead to the `BUG()` condition, as `ReiserFS` is in maintenance mode.

Note that there are other notable efforts on finding file system bugs through fuzzing or manual auditing.

- Syzkaller, the state-of-the-art system call fuzzer that started to support mutating file system images in March, 2018. Note that Google deployed many more instances of Syzkaller (i.e., syzbot) than those of JANUS for continuously fuzzing the upstream kernel. Although syzbot fuzzes the whole kernel, we found more file system bugs with JANUS in four months. According to our investigation, Syzkaller reported only two `ext4` bugs, one `XFS` bug, four `F2FS` bugs, and one `HFS+` bug during our evaluation period, among which one of the `ext4` bugs, the `XFS` bug, and the `HFS+` bug were also found by JANUS. JANUS missed one `ext4` bug requiring a 4K block size, which is larger than that of our seed images. And we started using JANUS to fuzz `F2FS` after these four `F2FS` bugs were fixed.
- Google Project Zero, a team of security researchers seeking zero-day vulnerabilities who found one `ext4` bug through source review. The bug was also discovered by JANUS.
- Internal efforts from the file system development community. XFS developers noticed four `XFS` bugs found by JANUS before we reported them. Unfortunately, we were unable to provide the total number of memory safety bugs found by developers whose patches cannot easily be differentiated from the ones for fixing functionalities.

Table VI lists the details of 43 patched bugs that were previously unknown. The bugs have a wide range of types, from relatively harmless floating point exceptions to critical out-of-bound access and heap overflow bugs that can be used to corrupt critical kernel data and execute arbitrary code with kernel privileges. Most of the bugs require mounting a corrupted image followed by particular file operations to trigger, which are the joint effects from two types of input of a file system that JANUS manages to explore. In particular, one needs to invoke three or more system calls to trigger 80% of these bugs, which indicates the effectiveness of the system call fuzzer. Moreover, a quarter of the bugs are triggered by mounting only a corrupted image, which further proves the effectiveness of JANUS in fuzzing images. As JANUS emphasizes the priority of mutating image bytes, all the generated test cases contain the images with error bytes. Therefore, no reported bug only requires particular file operations without an uncorrupted image to trigger.

Result. JANUS successfully found 90 bugs in widely-used and mature file systems in upstream kernels. Among them, 62 bugs have been confirmed as previously unknown. As a specialized fuzzer for file systems, JANUS helped the Linux kernel community to discover and patch more file system bugs than Syzkaller in recent months.

B. Exploring the State Space of Images

We first evaluate how JANUS mutates image bytes to explore a target file system by comparing it with Syzkaller. Syzkaller recently supported mounting mutated images by introducing a wrapper call: `syz_mount_image()`, which takes mutated non-zero segments of an image as input, flushes them into a loop device at corresponding offsets, and eventually invokes `mount()`. To evaluate the impact of state space of an input image, we disable the system call fuzzing stage in JANUS and concentrate only on fuzzing the image. We denote our image fuzzer as JANUS_i. After a mutated image is mounted, we enforce both our LKL-based executor, used by JANUS_i, and the executor of Syzkaller (called $\text{Syzkaller}_\text{i}$) running in a KVM instance to perform a fixed sequence of system calls under the mounting point (see Figure 13) to demonstrate how mutated image bytes help fuzzers to explore a file system. We evaluate both fuzzers, with the seed images of eight file systems, described in the experimental setup, for 12 hours. For each target file system, we launch one JANUS$_i$ instance and one KVM instance for Syzkaller$_i$.

Figure 8 presents the number of paths both JANUS$_i$ and Syzkaller$_i$ visit in selected file systems. After running for 30 minutes, JANUS$_i$ always has higher code coverage than Syzkaller$_i$. JANUS$_i$ outperforms Syzkaller$_i$ by 1.47–4.17× for the evaluated file systems. Note that most selected file systems have relatively complex implementation, which shows the ability of JANUS mutating important image bytes to discover deeper code paths. Our approach differs from Syzkaller$_i$, as Syzkaller$_i$ considers only the important parts of an image as an array of non-zero chunks, that can either miss metadata blocks or even include inessential data blocks. By contrast, JANUS$_i$ leverages the semantics of an image, namely locating and mutating metadata blocks only. In addition, both `GFS2` and `Btrfs` have checksum for metadata blocks, which severely degrades the performance of Syzkaller$_i$. Another interesting observation is that Syzkaller$_i$ does not correctly use the seed image for `XFS` because Syzkaller does not support an image containing more than 4096 non-contiguous non-zero chunks, which is one of the big limitations of Syzkaller in fuzzing file systems. Therefore, Syzkaller$_i$ has to generate `XFS` images from scratch. Since `XFS` v5 has metadata checksum, Syzkaller$_i$ cannot make any forward progress even after running for 12 hours, as it does not fix the checksum of metadata.

Result. By mutating metadata blocks and fixing checksums, JANUS$_i$ quickly explores more code paths in the selected file systems than Syzkaller$_i$ when fuzzing only images with fixed file operations. More specifically, JANUS$_i$ achieves at most 4.17× more code coverage than Syzkaller$_i$, which shows the effectiveness of JANUS when fuzzing only images.

C. Exploring File Operations

We now evaluate the effectiveness of only fuzzing file operations without mutating the file system image, i.e., we discard the image fuzzing stage. We denote our file operation fuzzer as JANUS$_s$, which automatically generates nine seed programs for mutation after inspecting a seed image, each one containing an `open()` system call on a file or directory in the image (see Figure 9). We compare JANUS$_s$ with Syzkaller$_s$ by fuzzing 27 file system-specific system calls[2] and executing generated programs on a seed image after being mounted. We hard-code the paths of all available files and directories on a seed image in the description file for Syzkaller$_s$ to fill the values of certain arguments when fuzzing particular system calls. We run both of these fuzzers on eight file systems for 12 hours.

[2] Syzkaller$_s$ and JANUS$_s$ fuzz the following system calls: `read()`, `write()`, `open()`, `lseek()`, `getdents64()`, `pread64()`, `pwrite64()`, `stat()`, `lstat()`, `rename()`, `fsync()`, `fdatasync()`, `access()`, `ftruncate()`, `truncate()`, `utimes()`, `mkdir()`, `rmdir()`, `link()`, `unlink()`, `symlink()`, `readlink()`, `chmod()`, `setxattr()`, `fallocate()`, `listxattr()` and `removexattr()`

As already mentioned, we launch one JANUS$_s$ instance and one KVM instance for Syzkaller$_s$ in this experiment. Further, we re-execute all programs generated by JANUS$_s$ to obtain comparable path coverage in KCOV style.

Figure 8 presents the evaluation result, which shows that with a wiser fuzzing strategy, JANUS$_s$ keeps exploring more code paths than Syzkaller$_s$ in the span of 12 hours. In particular, JANUS$_s$ eventually visits 2.24×, 1.27×, and 1.25× more unique code paths than Syzkaller$_s$ when fuzzing the three most popular file systems, `XFS` v5, `Btrfs`, and `ext4`, respectively. Moreover, JANUS$_s$ also outperforms Syzkaller$_s$ 1.72× and 1.49× on `HFS+` and `GFS2`, respectively. By generating context-aware workloads, we observe that JANUS is more effective than Syzkaller$_s$ for fuzzing file systems. The reason is that Syzkaller$_s$ is a general and advanced system call fuzzer, but, unlike JANUS$_s$, Syzkaller$_s$ completely fails to exploit the domain knowledge of a file system to explore its code path effectively.

Result. By generating context-aware workloads, JANUS explores more code paths than Syzkaller$_s$ in all eight popular file systems when only targeting the system calls related to file operations. In particular, the programs generated by JANUS manage to visit at most 2.24× more paths. The evaluation result fully demonstrates the effectiveness of JANUS in terms of file operation fuzzing.

D. Exploring Two-Dimensional Input Space

To demonstrate the comprehensiveness of JANUS'S fuzzing by mutating both image bytes and file operations, we run original JANUS and Syzkaller on the eight aforementioned file systems with the same seed images for 12 hours. We provide `syz_mount_image()` in the description file to make Syzkaller not only generate system calls but also mutate the bytes in a seed image while invoking 27 file system-specific system calls (see §V-C). In this experiment, we simultaneously launch three instances for both JANUS and Syzkaller for parallel fuzzing. Moreover, both fuzzers share generated test cases for each corresponding file systems. Figure 8 (marked Syzkaller and JANUS) shows the results of this experiment.

We observe that JANUS discovers more code paths than both JANUS$_i$ and JANUS$_s$. Our results illustrate the importance of fuzzing both images and file operations to comprehensively explore a file system. More important, JANUS further outperforms Syzkaller on all tested file systems. In particular, JANUS achieves at most 4.19×, 4.04×, and 3.11× higher code coverage than Syzkaller when fuzzing `Btrfs`, `GFS2`, and `F2FS`, respectively. For `ext4`, JANUS also hits 2.01× more unique code paths. The major reason is that Syzkaller prioritizes system call fuzzing over image fuzzing, while JANUS incorporates the strategy of *blob-directed system call* fuzzing. For instance, while generating a program for fuzzing, Syzkaller does not guarantee whether a valid file system is mounted before performing any file operation, i.e., it completely forgoes the file system context-awareness to blindly fuzz a file system. We mitigate this issue by invoking `umount()` and `syz_mount_image()` at the beginning of a program. Nevertheless, Syzkaller is still not capable of stopping if mounting a mutated image fails. In

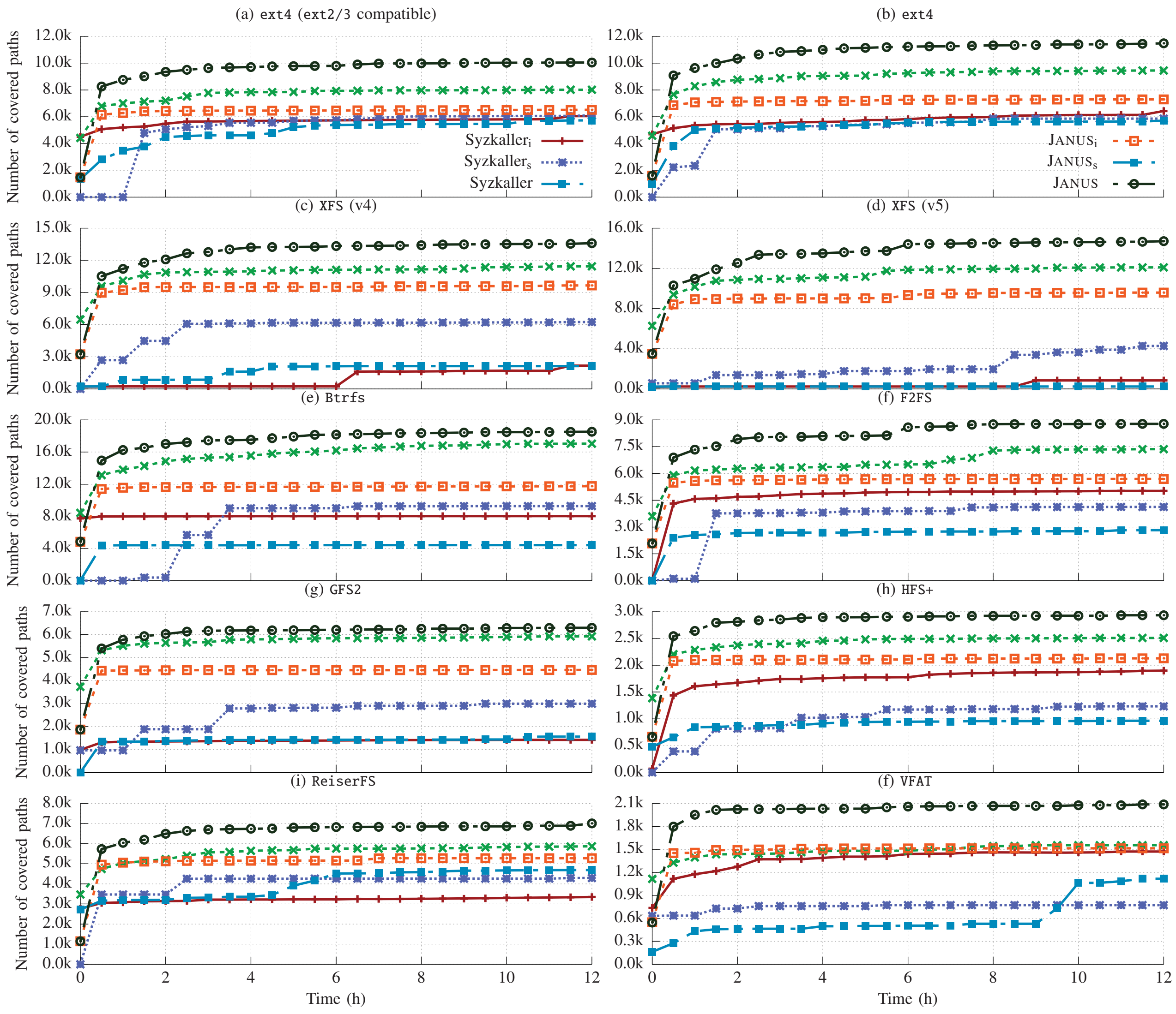

Fig. 8: The overall path coverage of using Syzkaller and JANUS to fuzz eight file system images for 12 hours. The y-axis represents the number of unique code paths of each file system visited during the fuzzing process. In particular, JANUS$_i$ and Syzkaller$_i$ only mutate bytes on a seed image and perform a fixed sequence of system calls on a mutated image and JANUS$_i$ outperforms Syzkaller$_i$ up to 4.17×. JANUS$_s$ and Syzkaller$_s$ generate random system calls to be executed on a fixed seed image, in which JANUS$_s$ achieves up to 2.24× higher coverage than Syzkaller$_s$. JANUS and Syzkaller fuzz both image bytes and file operations, and JANUS visits at most 4.19× unique paths.

fact, `syz_mount_image()` can be invoked anywhere and several times in a program generated by Syzkaller. Unlike Syzkaller, JANUS fuzzes each image separately with a clean LKL instance. If mounting a mutated image succeeds, the executor will execute context-aware workloads afterward and terminate with `umount()`. As we mentioned in §V-B, the comparison for `XFS` is partially unfair due to the limitation of Syzkaller in handling dense images. Another advantage of JANUS is that it utilizes many fewer CPU and memory resources for LKL instances but still outperforms Syzkaller, which relies on VMs.

Result. JANUS achieves higher code coverage than both JANUS$_s$ and JANUS$_i$, which proves the importance of mutating both images and operations in file system fuzzing. Moreover, JANUS outperforms Syzkaller on all eight file systems. In particular, JANUS outperforms Syzkaller at most 4.19× on `Btrfs`, one of the popular file systems that has an extremely complex design. Our evaluation result shows the effectiveness of JANUS in fuzzing a file system by exploring its two-dimensional input space.

E. Reproducing Crashes

To evaluate whether the library OS used by JANUS (i.e., LKL) helps to reproduce more found crashes compared to VMs, for each collected crashing input generated in the final experiment where both of two types of inputs are mutated (see §V-D), we first use our PoC generator to parse out the image

File System	Syzkaller	JANUS	#Unique
ext4 (com.)	0/7 (0%)	16/16 (100%)	6
ext4	0/3 (0%)	196/196 (100%)	8
XFS v4	0/2517 (0%)	24/24 (100%)	2
XFS v5	0/6 (0%)	67/67 (100%)	2
Btrfs	0/0 (0%)	1793/2054 (88%)	18
F2FS	0/1288 (0%)	2390/2458 (97%)	28
GFS2	0/916 (0%)	1030/1080 (95%)	12
HFS+	0/8 (0%)	815/815 (100%)	6
ReiserFS	0/2535 (0%)	1800/1800 (100%)	20
VFAT	0/0 (-)	0/0 (-)	0

TABLE IV: The bug reproducibility of Syzkaller and JANUS using KVM instances and LKL-based executors, respectively. For each **X/Y** pair in the table, **X** indicates the number of crashes triggered by a fuzzer during our experiment in §V-D, and **Y** represents the number of crashes that can be reproduced again with saved crashing inputs. The column **#Unique** reports the unique crashes among the ones found by JANUS in the experiment based on their crashing `PC` values.

Reboot VM	Revert snapshot	LKL
14.5s	1.4s	10.7ms

TABLE V: The average time costs of VM-based (i.e., KVM) fuzzer and JANUS for a non-aging OS and file system. The total time includes reloading a clean-slate OS and mounting an image.

and a particular sequence of system calls. We then mount the image and execute system calls under the mounting point again to see if the kernel crashes. Based on the crashing `PC` values, we also count the number of unique crashes among those reproducible ones. Table IV summarizes the number of crashes and reproducible ones found by JANUS and Syzkaller. Note that Syzkaller originally records these numbers in its logs. Because of the fundamental limitation of using an aging OS, in which Syzkaller mounts different images and invokes system calls without initialization, Syzkaller fails to reproduce any of its found crashes. On the contrary, JANUS can reproduce more than 95% of crashes found in most file systems, except `Btrfs`. `Btrfs` launches multiple kernel threads completing different transactions in parallel, which results in non-deterministic kernel execution. In addition, `F2FS` and `GFS2` also spawns few worker threads to accomplish particular tasks, such as garbage collection, logging, etc. Note that, in theory, it is possible to reproduce 100% of crashes if we can control the thread scheduling, which is currently outside the scope of this work.

We also estimate the performance overhead of bringing up a fresh copy of OS (non-aging OS) for a VM-based fuzzer to test every generated input. More specifically, we evaluate the total time that a KVM instance (two cores and 2GB memory) spends on either rebooting VM or reverting an existing snapshot and testing an input image, and compare it to the corresponding time that our LKL executor requires. Table V presents the evaluation result. By simply invoking `fork()` to launch a new LKL instance, our LKL-based executor spends negligible time on setting up a clean OS and a fresh file system compared with a KVM instance.

Result. LKL, on which JANUS relies, provides a clean-slate OS that has more stable execution than an OS running in a VM. This approach results in reproducing most of the crashes. In particular, JANUS is able to reproduce at least 88% of the crashes found during a 12-hour fuzzing period. By contrast, a VM-based fuzzer (i.e., Syzkaller) fails to reproduce any of its crashes. Moreover, re-initializing OS states in a VM suffers from unacceptable overhead.

F. *Miscellany*

Besides finding previously unknown bugs, JANUS contributes the following notable results to the file system development community.

Malicious image samples. The development communities of several file systems including `Btrfs`, `F2FS`, etc., have already added a number of corrupted images generated by JANUS into their repositories for internal fuzzing and for future regression testing. Currently, developers consider these images as representative malicious samples that involve diverse error bytes in various metadata fields for testing the functionality of file systems.

General patches for file system hardening. `F2FS` developers have not only fixed the bugs reported by us in the kernel module but also extended corresponding security checks into the user-space tool (i.e., `fsck.f2fs`) to help users detect these image corruptions in advance, i.e., before the Linux kernel mounts images containing critical error bytes.

VI. DISCUSSION

We have demonstrated that JANUS effectively explores the code paths and discovers unknown bugs in a disk file system in the Linux kernel. We now discuss the limitations of JANUS and our future directions.

Library OS based executor. JANUS relies on LKL to test in-kernel file systems. In fact, other OS fuzzers can use it to test other kernel sub-systems, except MMU-dependent components. For instance, JANUS cannot fuzz the DAX mode of a file system [31] without modification on LKL. We could also use user-mode Linux (UML), as done by Oracle's kernel fuzzer [48]. However, UML suffers from the limitation of its multi-process design, which complicates the spotting of a kernel crash and termination of all its processes during each iteration. Therefore, UML does not support fuzzing the kernel as a user application well.

Minimal PoC generator. An ideal PoC for developers to debug crashes consists of an image that only has essential error bytes and a program with the least file operations. To achieve this, JANUS currently uses a brute force approach to revert every mutated byte and also tries to remove every invoked file operation to check whether the kernel still crashes at the expected location. Although this approach is sub-optimal, we can leverage certain file system utilities such as `fsck` and `debugfs` and system call trace distillation techniques [22, 49] to pinpoint root-causing bytes and system calls. Another possibility is to apply taint tracking on the kernel.

Fuzzing FUSE drivers. Currently, JANUS does not support file systems (e.g., NTFS [70], GVfs [67], SSHFS [55], etc.)

that rely on FUSE (Filesystem in Userspace) [32]. We can easily extend the fuzzing engine of JANUS to fuzz such file systems as long as they store user data in a disk image and support certain file operations for users to interact with data.

Fuzzing file system utilities. Developers heavily rely on system utilities (e.g., `mkfs`, `fsck`, etc.), to manage file systems. For instance, Linux automatically launches `fsck` for recovering disk data from a sudden system crash. Moreover, users use `fsck` to check the consistency of an untrusted disk image before mounting the disk. Hence, developers desire such utilities to be bug free. We believe that developers can easily extend the image mutator of JANUS to generate corrupted images for fuzzing these tools, thereby improving their robustness. In fact, we use JANUS to find two unknown bugs in `fsck.ext4`, and one has already been fixed.

Extending to fuzz file systems on other OSes. Extending JANUS for fuzzing in-kernel file systems on other OSes will be straightforward if the corresponding library OS solution exists. For instance, Drawbridge [52] enables Windows to efficiently run in a process. Moreover, we can also integrate the core fuzzing engine of JANUS with other general kernel fuzzing frameworks such as kAFL [61] built upon QEMU and KVM to fuzz file systems used by other commodity OSes such as Windows and macOS.

Improving other file system testing tools. The goal of JANUS is to find general security bugs in file systems, contrary to the goals of other tools, including crash-consistency checkers [6, 73] and semantic correctness checkers [36, 58]. However, these tools also need sequences of file operations. Hence, JANUS becomes a one-stop solution on which other tools can rely.

VII. Related work

Structured input fuzzing. Numerous approaches have been proposed to fuzz inputs that are highly structured like file system images. Unlike JANUS, a number of generation fuzzers ([14, 23, 41, 42, 50]) construct syntactically correct inputs from scratch based on input specifications described through manual efforts. Furthermore, EXE [7] relies on symbolic execution to build valid inputs that satisfy deep path constraints. More advanced approaches such as [3, 16, 24] learn the input structures from a set of samples. On the other side, mutation-based fuzzers [4, 5, 10, 15, 18, 34, 76] generate new inputs by mutating valid samples. The generated inputs have correct structures with slight errors, and hopefully trigger bugs. Considering the complexity of a file system image and the diversity in image format among different file systems, JANUS adopts mutation-based strategies to fuzz images. Similar to file system images, many file formats involve checksums for integrity checks. JANUS specifically fixes metadata checksums with expertise knowledge. Nevertheless, some checksum-aware fuzzers [33, 71] identify checksum fields and bypass checksum checks at runtime through dynamic taint analysis.

OS kernel fuzzers. To find security bugs in OSes, a number of general kernel fuzzing frameworks [20, 43, 46, 61] and OS-specific kernel fuzzers [22, 25, 44, 45, 47] have been proposed. Unlike JANUS, all these fuzzers generate random system calls based upon predefined grammar rules, which is ineffective in the context of file system fuzzing. Several recent OS fuzzers such as IMF [22] and MoonShine [49] focusing on seed distillation are orthogonal to this work. Nevertheless, JANUS can start with seed programs of high quality by utilizing their approaches.

File system semantic correctness checkers. JUXTA [36] and SibylFS [58] are other types of file system checkers, that aim to find whether the implementation of a file system exactly meets the standard (e.g., the POSIX standard, man pages, etc.) through static analysis and high-level modeling of file system behaviors. They are orthogonal to JANUS regarding their purposes and methodologies. Similarly, JANUS can generate meaningful system calls to find crash consistency bugs [6, 73].

File system abstraction. Several studies [65, 66] propose general interfaces for file system utilities to access and manipulate the on-disk metadata of various file systems through high-level abstraction. By utilizing these interfaces, JANUS can compress disk images in a more general manner without implementing an image parser for every target file system.

VIII. Conclusion

In this work, we propose JANUS, an evolutionary file system fuzzer, that explores an in-kernel file system by exploring its two-dimensional input space (i.e., images and file operations). Unlike existing file system fuzzers, JANUS efficiently mutates metadata blocks of input images while emitting context-aware workloads on an image. Rather than traditional VMs, JANUS relies on a library OS that supports fast reloading to test OS functionalities, thereby avoiding unstable executions and irreproducible bugs. We reported 90 bugs found by JANUS in the upstream kernel, 43 of which have been fixed with 32 CVEs assigned. JANUS outperforms Syzkaller by exploring at most 4.19× more code paths when fuzzing popular file systems for 12 hours and manages to reproduce 88–100% of found crashes. We will open source our implementation of JANUS, which has been requested by several file system development communities due to our notable results. We believe that JANUS will be one-stop solution for file system testing, as JANUS can act as a basic infrastructure to design new semantic and crash-consistency checkers for file systems.

IX. Acknowledgment

We thank the anonymous reviewers, and our shepherd, Thorsten Holz, for their helpful feedback. We also thank all the file system developers, including Theodore Ts'o, Darrick J. Wong, Dave Chinner, Eric Sandeen, Chao Yu, Wenruo Qu and Ernesto A. Fernández for handling our bug reports. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N000141512162, DARPA TC (No. DARPA FA8650-15-C-7556), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-year Study of File-system Metadata. In *Proceedings of the ACM Transactions on Storage (TOS)*, 2007.

[2] Apple Inc. macOS High Sierra. https://www.apple.com/macos/high-sierra, 2018.

[3] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, pages 95–110. ACM, 2017.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[6] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[8] M. Cao, S. Bhattacharya, and T. Ts'o. Ext4: The next generation of ext2/3 filesystem. In *USENIX Linux Storage and Filesystem Workshop*, 2007.

[9] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying Crash Safety for Storage Systems. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015.

[10] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[11] J. Corbet. Improving ext4: bigalloc, inline data, and metadata checksums. https://lwn.net/Articles/469805, 2011.

[12] J. Corbet. Filesystem mounts in user namespaces. https://lwn.net/Articles/652468, 2015.

[13] J. Dike. User-mode Linux. In *Annual Linux Showcase Conference*, 2001.

[14] I. Fratric. DOM fuzzer. https://github.com/googleprojectzero/domato, 2018.

[15] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[16] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Champaign, IL, Oct. 2017.

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

[18] Google. OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2018.

[19] Google. syzbot. https://syzkaller.appspot.com, 2018.

[20] Google. syzkaller is an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2018.

[21] S. Grubb. fsfuzzer-0.7. http://people.redhat.com/sgrubb/files/fsfuzzer-0.7.tar.gz, 2009.

[22] H. Han and S. K. Cha. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[23] R. Hodován, Á. Kiss, and T. Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 45–48. ACM, 2018.

[24] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Champaign, IL, Oct. 2017.

[25] D. Jones. Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2018.

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

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

[28] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2011.

[29] M. Larabel. F2FS File-System Moves Forward With Encryption Support. https://www.phoronix.com/scan.php?page=news_item&px=F2FS-Encryption-Support, 2015.

[30] C. Lee, D. Sim, J. Y. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2015.

[31] Linux. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt, 2015.

[32] Linux. fuse - Filesystem in Userspace (FUSE) device. http://man7.org/linux/man-pages/man4/fuse.4.html, 2015.

[33] X. Liu, Q. Wei, Q. Wang, Z. Zhao, and Z. Yin. CAFA: A Checksum-Aware Fuzzing Assistant Tool for Coverage Improvement. *Security and Communication Networks*, 2018, 2018.

[34] LLVM Project. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2018.

[35] Microsoft. Windows. https://www.microsoft.com/en-us/windows, 2018.

[36] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

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

[38] MITRE Corporation. CVE-2017-13830. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13830, 2017.

[39] MITRE Corporation. CVE-2017-6990. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6990, 2017.

[40] MITRE Corporation. F2FS CVE entries. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=f2fs, 2018.

[41] Mozilla Corporation. MozPeach. https://github.com/MozillaSecurity/peach, 2017.

[42] Mozilla Corporation. JavaScript engine fuzzers. https://github.com/MozillaSecurity/funfuzz, 2018.

[43] MWR Labs. Cross Platform Kernel Fuzzer Framework. https://github.com/mwrlabs/KernelFuzzer, 2016.

[44] MWR Labs. macOS Kernel Fuzzer. https://github.com/mwrlabs/OSXFuzz, 2017.

[45] NCC Group. System call fuzzing of OpenBSD amd64 using TriforceAFL. https://github.com/nccgroup/TriforceOpenBSDFuzzer, 2016.

[46] NCC Group. AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL, 2017.

[47] NCC Group. A linux system call fuzzer using TriforceAFL. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2017.

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

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

[50] Peach Tech. Peach Fuzzer. https://sourceforge.net/projects/peachfuzz, 2016.

[51] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[52] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *ACM SIGPLAN Notices*, 2011.

[53] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*. ACM, 2011.

[54] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux kernel library. In *Proceedings of the 9th Roedunet International Conference (RoEduNet)*. IEEE, 2010.

[55] N. Rath and M. Szeredi. A network filesystem client to connect to SSH servers. https://github.com/libfuse/sshfs, 2018.

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

[57] Ribose Inc. FuzzBSD, a filesystem image fuzzing script to test BSD kernels. https://github.com/riboseinc/fuzzbsd, 2017.

[58] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[59] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. In *Proceedings of the ACM Transactions on Storage (TOS)*, 2013.

[60] B. Schneier. "Evil Maid" Attacks on Encrypted Hard Drives. https://www.schneier.com/blog/archives/2009/10/evil_maid_attac.html, 2009.

[61] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

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

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

[64] Silicon Graphics Inc. (SGI) and Red Hat Inc. XFS. http://xfs.org, 2018.

[65] K. Sun, D. Fryer, J. Chu, M. Lakier, A. D. Brown, and A. Goel. Spiffy: enabling file-system aware storage applications. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, CA, Feb. 2018.

[66] K. Sun, M. Lakier, A. D. Brown, and A. Goel. Breaking Apart the VFS for Managing File Systems. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*, Boston, MA, July 2018.

[67] The GNOME Project. GVfs. https://wiki.gnome.org/Projects/gvfs, 2018.

[68] L. Torvalds. Linux kernel source tree. https://github.com/torvalds/linux, 2018.

[69] T. Ts'o. Ext2/3/4 file system utilities. https://github.com/tytso/e2fsprogs, 2018.

[70] Tuxera. NTFS-3G. https://www.tuxera.com/community/open-source-ntfs-3g, 2017.

[71] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[72] M. Xie and L. Zefan. Performance improvement of btrfs. *LinuxCon Japan*, 2011.

[73] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[74] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the ACM Transactions on Computer Systems (TOCS)*, 2006.

[75] M. Zalewski. american fuzzy lop (2.52b) - config.h. https://github.com/mirrorer/afl/blob/master/config.h, 2017.

[76] M. Zalewski. american fuzzy lop (2.52b). http://lcamtuf.coredump.cx/afl, 2018.

[77] Zero Day Initiative. CVE-2018-4268. https://www.zerodayinitiative.com/advisories/ZDI-18-602, 2018.

Appendix

A. Seed Image

```
./               # root
./foo            # folder
./foo/bar        # folder
./foo/bar/acl    # file protected by ACL
./foo/bar/baz    # normal file
./foo/bar/fifo   # FIFO file
./foo/bar/hln    # hardlink to baz
./foo/bar/sln    # softlink to baz
./foo/bar/xattr  # file with an extended attribute
```

Fig. 9: The hierarchy of a seed image tested by JANUS in the evaluation.

Figure 9 presents the organization of files and directories stored on a seed image in our evaluation.

B. Coverage Profiling: AFL versus KCOV

```
/* AFL */
cur_location = <COMPILE_TIME_RANDOM_NUMBER>;
bitmap[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;

/* Syzkaller */
uint32_t pc = cover_data[i];
uint32_t sig = pc ^ prev;
prev = hash(pc);
```

Fig. 10: The code injected by AFL and the code used by Syzkaller to profile runtime path coverage.

AFL and KCOV used to support Syzkaller apply two different approaches to instrument a fuzzing target and reserve runtime path coverage in two different formats. In particular, AFL labels every basic block with a random number, and at each branch, the code shown in Figure 10 is instrumented. Each byte set in the bitmap can be considered as a hit on a particular code path.

KCOV relies on the `-fsanitize-coverage=trace-pc` flag of GCC (>= 6.0) to inject code at every basic block which emits the current `PC` value into a buffer mapped in user space. After the execution for a mutated input is completed, Syzkaller uses every two consequent `PC` values to calculate out a hash value to represent a particular code path (see Figure 10).

Note that Syzkaller uses the lowest 32 bits of a `PC` value to label the corresponding basic block, which has lower randomness compared to pseudo random numbers generated by AFL and thereby results in more collisions that degrade the fuzzing performance.

C. An LKL Example

Figure 11 provides a simple example of leveraging LKL in an user application to operate an `ext4` image.

D. Serialization Format

JANUS serializes a generate program along with the speculated image status into a binary file by following the format described in Figure 12.

E. A Testing Program for Image Fuzzing

Figure 13 presents a fixed sequence of system calls to be performed on any mutated image when evaluating the effectiveness of JANUS and Syzkaller in image fuzzing.

F. Supported System Calls

In our implementation, JANUS supports generating and mutating the following 34 system calls: `read()`, `write()`, `open()`, `seek()`, `mmap()`, `getdents64()`, `pread64()`, `pwrite64()`, `stat()`, `lstat()`, `rename()`, `fsync()`, `fdatasync()`, `syncfs()`, `sendfile()`, `access()`, `ftruncate()`, `truncate()`, `fstat()`, `statfs()`, `fstatfs()`, `utimes()`, `mkdir()`, `rmdir()`, `link()`, `unlink()`, `symlink()`, `readlink()`, `chmod()`, `fchmod()`, `setxattr()`, `fallocate()`, `listxattr()`, and `removexattr()`.

G. File system bugs found by JANUS

Table VI lists the patched bugs found by JANUS in five widely used file systems that were previously unknown.

#	File system	CVE	File	Function	Type	Conditions
1	ext4	CVE-2018-1092	fs/ext4/inode.c	ext4_iget	Use-after-free	I
2		CVE-2018-1093	fs/ext4/balloc.c	ext4_valid_block_bitmap	Out-of-bounds access	I+S
3		CVE-2018-1094	fs/ext4/super.c	ext4_fill_super	Null pointer dereference	I+S
4		CVE-2018-1095	fs/ext4/xattr.c	ext4_xattr_check_entries	Out-of-bounds access	I+S
5		CVE-2018-10840	fs/ext4/xattr.c	ext4_xattr_set_entry	Heap overflow	I+S
6		CVE-2018-10876	fs/ext4/extents.c	ext4_ext_remove_space	Use-after-free	I+S
7		CVE-2018-10877	fs/ext4/extents.c	ext4_ext_drop_refs	Out-of-bounds access	I+S
8		CVE-2018-10878	fs/ext4/balloc.c	ext4_init_block_bitmap	Out-of-bounds access	I+S
9		CVE-2018-10879	fs/ext4/xattr.c	ext4_xattr_set_entry	Use-after-free	I+S
10		CVE-2018-10880	fs/ext4/inline.c	ext4_update_inline_data	Out-of-bounds access	I+S
11		CVE-2018-10881	fs/ext4/ext4.h	ext4_get_group_info	Uninitialized memory	I+S
12		CVE-2018-10882	fs/jbd2/transaction.c	start_this_handle	BUG()	I+S
13		CVE-2018-10883	fs/jbd2/transaction.c	jbd2_journal_dirty_metadata	BUG()	I+S
14		-	fs/ext4/xattr.c	ext4_xattr_set_entry	Heap overflow	I+S
15		-	fs/ext4/namei.c	ext4_rename	Use-after-free	I+S
16		-	fs/ext4/inline.c	empty_inline_dir	Divide by zero	I+S
17	XFS	CVE-2018-13093	fs/xfs/xfs_icache.c	xfs_iget_cache_hit	Use-after-free	I+S
18		CVE-2018-10322	fs/xfs/xfs_inode.c	xfs_ilock_attr_map_shared	Null pointer dereference	I+S
19		CVE-2018-10323	fs/xfs/libxfs/xfs_bmap.c	xfs_bmapi_write	Null pointer dereference	I+S
20		CVE-2018-13094	fs/xfs/xfs_trans_buf.c	xfs_trans_binval	Null pointer dereference	I+S
21		CVE-2018-13095	fs/xfs/libxfs/xfs_bmap.c	xfs_bmap_extents_to_btree	Out-of-bounds access	I+S
22		-	fs/xfs/libxfs/xfs_alloc.c	xfs_alloc_get_freelist	Null pointer dereference	I+S
23		-	fs/xfs/libxfs/xfs_dir2.c	xfs_dir_isempty	Null pointer dereference	I+S
24	Btrfs	CVE-2018-14609	fs/btrfs/relocation.c	__del_reloc_root	Null pointer dereference	I
25		CVE-2018-14610	fs/btrfs/extent_io.c	write_extent_buffer	Out-of-bounds access	I+S
26		CVE-2018-14611	fs/btrfs/free-space-cache.c	try_merge_free_space	Use-after-free	I
27		CVE-2018-14612	fs/btrfs/ctree.c	btrfs_root_node	Null pointer dereference	I
28		CVE-2018-14613	fs/btrfs/free-space-cache.c	io_ctl_map_page	Null pointer dereference	I+S
29		-	fs/btrfs/volumes.c	btrfs_free_dev_extent	BUG()	I+S
30		-	fs/btrfs/locking.c	btrfs_tree_lock	Deadlock	I
31		-	fs/btrfs/volumes.c	read_one_chunk	BUG()	I
32	F2FS	CVE-2018-13096	fs/f2fs/segment.c	build_sit_info	Heap overflow	I
33		CVE-2018-13097	fs/f2fs/segment.h	utilization	Divide by zero	I
34		CVE-2018-13098	fs/f2fs/inode.c	f2fs_iget	Out-of-bounds access	I+S
35		-	fs/f2fs/segment.h	verify_block_addr	BUG()	I+S
36		CVE-2018-13099	fs/f2fs/segment.c	update_sit_entry	Use-after-free	I+S
37		CVE-2018-13100	fs/f2fs/segment.c	reset_curseg	Divide by zero	I
38		-	fs/inode.c	clear_inode	BUG()	I
39		-	fs/f2fs/node.c	f2fs_truncate_inode_blocks	BUG()	I+S
40		CVE-2018-14614	fs/f2fs/segment.c	__remove_dirty_segment	Out-of-bounds access	I
41		CVE-2018-14615	fs/f2fs/inline.c	f2fs_truncate_inline_inode	Heap overflow	I+S
42		CVE-2018-14616	fs/crypto/crypto.c	fscrypt_do_page_crypto	Null pointer dereference	I+S
43	HFS+	CVE-2018-14617	fs/hfsplus/dir.c	hfsplus_lookup	Null pointer dereference	I+S

TABLE VI: The list of previously unknown bugs in widely used file systems found by JANUS that have already been fixed in Linux kernel v4.16, v4.17, and v4.18. We are still waiting for CVE assignment for several confirmed bugs. For security concerns, we exclude other 19 found bugs that developers have not fixed. The rightmost column, **Conditions**, indicates what components of JANUS contribute to discovering the bugs. **I** means that triggering the bug only requires mounting a mutated image. **I+S** represents that the bug is triggered by mounting a mutated image and also invoking specific system calls.

```c
int mount_and_read(char *fsimg_path) {
    struct lkl_disk disk;
    char mpoint[32], buffer[1024];
    unsigned int disk_id;
    char *file;
    int fd;
    disk.fd = open(fsimg_path, O_RDWR);
    disk.ops = NULL;
    disk_id = lkl_disk_add(&disk);
    lkl_start_kernel(&lkl_host_ops, "mem=128M");
    lkl_mount_dev(disk_id, 0, "ext4", 0,
                "errors=remount-ro", mpoint, sizeof(mpoint));
    asprintf(&file, "%s/file", mpoint);
    fd = lkl_sys_open(file, LKL_O_RDONLY, 0666);
    if (fd >= 0) {
        lkl_sys_read(fd, buf, 1024);
        lkl_sys_close(fd);
    }
    lkl_umount_dev(disk_id, cla.part, 0, 1000);
    lkl_disk_remove(disk);
    lkl_sys_halt();
}
```

Fig. 11: A function example that mounts an `ext4` image and reads a file stored on the image through LKL APIs.

```protobuf
message Variable {
    required int32 index; // variable index
    required int32 size; // variable size
    required bool is_pointer; // if the variable is a pointer
    // the buffer data pointed to by a pointer
    required bytes buffer;
    // the file object type of an active file descriptor
    // for normal variables, it is -1
    required int32 type;
}

message Variables {
    repeated Variable variables;
}

message Arg {
    // if the argument is a variable
    required bool is_var;
    // an immediate value or
    // the index of the corresponding variable
    required int64 value;
}

message Syscall {
    required int32 nr; // syscall number
    repeated Arg args;
    // the index of the variable that
    // stores the return value of the syscall
    // if necessary (e.g., fd returned from open());
    // by default it is -1
    required int64 ret_index;
}

message FileObject {
    required string path; // relative path
    // the file object type (FILE, DIR, SYMLINK, etc.)
    required int32 type;
    // the names of all the extended attributes
    repeated string xattr_names;
}

message Program {
    repeated Syscall syscalls;
}

message Status {
    repeated FileObject fobjs;
}
```

Fig. 12: The format of a serialized program and speculated image status described in protocol buffer language.

```c
void activity(const char *mountpoint)
{
    DIR *dir = opendir(mountpoint);
    if (dir) {
        readdir(dir);
        closedir(dir);
    }
    static int buf[8192];
    memset(buf, 0, sizeof(buf));
    int fd = open(foo_bar_baz, O_RDONLY);
    if (fd != -1) {
        void *mem = mmap(NULL, 4096, PROT_READ,
                    MAP_PRIVATE | MAP_POPULATE, fd, 0);
        munmap(mem, 4096);
        read(fd, buf, 11);
        read(fd, buf, sizeof(buf));
        close(fd);
    }
    fd = open(foo_bar_baz, O_RDWR | O_TRUNC, 0777);
    if (fd != -1) {
        write(fd, buf, 517);
        write(fd, buf, sizeof(buf));
        fdatasync(fd);
        fsync(fd);

        lseek(fd, 0, SEEK_SET);
        read(fd, buf, sizeof(buf));
        lseek(fd, 1234, SEEK_SET);
        read(fd, buf, 517);
        close(fd);
    }
    fd = open(foo_bar_baz, O_RDWR | O_TRUNC, 0777);
    if (fd != -1) {
        lseek(fd, 1024 - 33, SEEK_SET);
        write(fd, buf, sizeof(buf));
        lseek(fd, 1024 * 1024 + 67, SEEK_SET);
        write(fd, buf, sizeof(buf));
        lseek(fd, 1024 * 1024 * 1024 - 113, SEEK_SET);
        write(fd, buf, sizeof(buf));
        lseek(fd, 0, SEEK_SET);
        write(fd, buf, sizeof(buf));
        fallocate(fd, 0, 0, 123871237);
        fallocate(fd, 0, -13123, 123);
        fallocate(fd, 0, 234234, -45897);
        fallocate(fd, FALLOC_FL_KEEP_SIZE |
                    FALLOC_FL_PUNCH_HOLE, 0, 4243261);
        fallocate(fd, FALLOC_FL_KEEP_SIZE |
                    FALLOC_FL_PUNCH_HOLE, -95713, 38447);
        fallocate(fd, FALLOC_FL_KEEP_SIZE |
                    FALLOC_FL_PUNCH_HOLE, 18237, -9173);
        close(fd);
    }
    rename(foo_bar_baz, foo_baz);
    struct stat stbuf;
    memset(&stbuf, 0, sizeof(stbuf));
    stat(foo_baz, &stbuf);
    chmod(foo_baz, 0000);
    chmod(foo_baz, 1777);
    chmod(foo_baz, 3777);
    chmod(foo_baz, 7777);
    chown(foo_baz, 0, 0);
    chown(foo_baz, 1, 1);
    unlink(foo_bar_baz);
    unlink(foo_baz);
    mknod(foo_baz, 0777, makedev(0, 0));
    char buf2[113];
    memset(buf2, 0, sizeof(buf2));
    listxattr(xattr, buf2, sizeof(buf2));
    removexattr(xattr, "user.mime_type");
    setxattr(xattr, "user.md5", buf2, sizeof(buf2), XATTR_CREATE);
    setxattr(xattr, "user.md5", buf2, sizeof(buf2), XATTR_REPLACE);
    readlink(sln, buf2, sizeof(buf2));
}
```

Fig. 13: The fixed file operations used for evaluating how effectively JANUS and Syzkaller fuzz images.