PAPER

File Systems are Hard to Test — Learning from Xfstests

Naohiro AOTA[†a)], *Nonmember and* **Kenji KONO**[†], *Member*

SUMMARY Modern file systems, such as ext4, btrfs, and XFS, are evolving and enable the introduction of new features to meet ever-changing demands and improve reliability. File system developers are struggling to eliminate all software bugs, but the operating system community points out that file systems are a hotbed of critical software bugs. This paper analyzes the code coverage of xfstests, a widely used suite of file system tests, on three major file systems (ext4, btrfs, and XFS). The coverage is 72.34%, and the uncovered code runs into 23,232 lines of code. To understand why the code coverage is low, the uncovered code is manually examined line by line. We identified three major causes, peculiar to file systems, that hinder higher coverage. First, covering all the features is difficult because each file system provides a wide variety of file-system specific features, and some features can be tested only on special storage devices. Second, covering all the execution paths is difficult because they depend on file system configurations and internal on-disk states. Finally, the code for maintaining backward-compatibility is executed only when a file system encounters old formats. Our findings will help file system developers improve the coverage of test suites and provide insights into fostering the development of new methodologies for testing file systems.

key words: *file system, code coverage, software test*

1. Introduction

File systems play a crucial role in achieving high reliability of computer systems. File systems store a boot loader, operating system binary, application binaries, and sensitive data that must not be lost or broken after power failures or unexpected system shutdown. If file systems are not reliable, the computer system cannot boot again, and the critical information stored there becomes permanently unavailable. To improve the reliability of file systems, modern file systems, such as ext4 [1], btrfs [2], and XFS [3], are evolving and enable the introduction of new features. In 2014, ext4, btrfs, and XFS developers made 775, 299, and 381 commits and modified 26,308, 9,038, and 109,993 lines of code in total, respectively.

Novel features added to modern file systems provide richer functionalities and potentially improve the reliability of file systems but complicate the design and implementation, sometimes resulting in buggy, unstable, unreliable file systems. File system developers are struggling to eliminate all software bugs, but the operating system community points out that file systems are a hotbed of critical software bugs [4]–[6]; almost 40% of Linux patches for file systems are for bug fixes [5]. In addition, file system bugs are very difficult to find; more than a half of bugs in file systems are not fixed for over a year [4]. For example, a data-loss fault in ext4, which is usually considered a mature and well-maintained file system, has not been fixed for at least two years [7].

To disclose bugs before release, file systems are tested using a suite of test cases as in other software systems. Residual bugs are considered to slip through test cases; thus, are not disclosed at test time. In this paper, we investigate the code coverage of xfstests [8], a popular suite of test cases for Linux file systems, and manually examine the *uncovered* code line by line. Xfstests was originally developed for XFS but is widely used and maintained in the Linux community. Our insight behind the code coverage analysis is that bugs undiscovered at test time are lurking in the code *not* covered in test cases. If the code coverage is improved, it is expected that more bugs can be disclosed at test time since higher code coverage generally induces higher fault coverage [9]. Thus, we expect, also in file-systems, that achieving higher code coverage of test cases results in higher fault coverage.

The code coverage of xfstests is measured on three file systems in this paper: ext4, btrfs, and XFS. We choose these file-systems because they are widely used, been actively developed in recent years, and used in many research activities. Ext4 is the de-facto standard in Linux and enabled by default on most Linux distributions. XFS is expected to be a next standard file system as it is the default file system on Red Hat Enterprise Linux 7. Btrfs is emerging as a next-generation file system. These three file systems have the top three commit numbers in local file systems. They all have more than 1,000 commits during Linux v3.x series. Lu et al. studied the evolution of these three file systems [5]. ReiserFS, JFS, and ext3 have also been studied, but ReiserFS and JFS have not been actively developed recently. Ext3 has already been merged into ext4. Code coverage of xfstests is 67.98%, 67.8%, and 81.86% in ext4, btrfs, and XFS, respectively. The total lines of code not covered in xfstests are 23,232 lines. The following conclusions are derived from the investigation:

File System Specific Tests: All the investigated file systems provide a wide variety of features specific to each file system. Covering all the features and all possible combinations of the features (or options) is challenging.

Manuscript received January 5, 2018.
Manuscript revised September 18, 2018.
Manuscript publicized November 7, 2018.

[†]The authors are with Keio University, Yokohama-shi, 223–8522 Japan.
a) E-mail: naota@sslab.ics.keio.ac.jp

DOI: 10.1587/transinf.2018EDP7006

Uncovered code due to not supporting file-system-specific features or options is 17%, 17% and 9% of uncoverage code in btrfs, ext4, and XFS, respectively.

Environmental Dependency: File systems are designed to run on various storage devices; thus, some features can be tested only on a special storage device. Therefore, it is necessary to prepare various storage devices and run test cases on each device to improve test coverage. This environmental dependency is dominant on btrfs (38% of the uncoverage) and accounts for 0.7% of the uncoverage in ext4 and 21% of that in XFS.

File System Configurations: File systems have a number of configurations, and some configurations change the execution paths of the same test case. For example, ext4 has about 60 options for mkfs and about 50 options for mount. Some of these options have inter-dependency; thus, it is not easy to enumerate all possible combinations. As a result, even if there is a test case for a certain feature, unexecuted code remains. We find 6%, 42% and 10% of the uncovered code is due to file system configurations for btrfs, ext4, and XFS, respectively.

On-Disk Structures: File systems build various data structures on storage devices to manage their files. Even if the same operation is conducted, the actual execution paths may differ depending on the state of on-disk structures. We find 15%, 10%, and 20% of the uncoverage in btrfs, ext4, and XFS are caused by the differences in on-disk structures.

File System Evolution: File systems sometimes update the formats of their on-disk structures when a new version is introduced. For example, ext4 updated the format of file-to-block mapping from ext2/3. File systems support reading/writing from/to such old formats. The code to read/write old formats is never executed unless we create and mount old-format file systems. This comprises 2%, 0.1%, and 1% of the uncoverage for btrfs, ext4, and XFS, respectively.

The rest of the paper is organized as follows. Section 2 describes xfstests and shows that the code coverage of the test is not high. Section 3 explores the causes of the code not being covered in test cases. Section 5 discusses related work, and Sect. 6 concludes the paper.

2. Low Coverage of Test Cases

A practical solution to reduce software bugs in file systems is to enhance the quality of test cases. If test cases catch most of the bugs at the test phase, the overall quality of file systems is expected to improve. In this section, we investigate the code coverage of xfstests, a test suite for Linux file systems.

2.1 Overview of Xfstests

Xfstests consists of three types of test cases: 1) generic, 2) shared, and 3) specific. *Generic* tests examine the fundamental features common to all file systems. *Shared* tests examine the advanced features supported by many, but not all, file systems. For example, the access control list (ACL) is not always supported by all file systems. *Specific* tests examine the features supported only by a specific file system. For example, btrfs supports the snapshot at the file-system level, which is usually supported outside the file system. Specific tests for btrfs contain test cases for the snapshot. The xfstests has 185 generic tests, 6 shared tests, and 304 specific tests (495 in total).

Each test case follows four steps: 1) check environmental requirements, 2) set up a file system if necessary, 3) run some commands, and 4) check the results. To enable the testing of creating and mounting file systems, two partitions must be prepared in xfstests before running test cases. One partition is called SCRATCH. A file system on the SCRATCH partition is created by a test case; thus, we can test the system calls for creating and mounting a file system. The other partition is called TEST. A file system on the TEST partition is created and mounted by testers, which allows the testers to tweak the options for creating and mounting a file system and initial content of the file system.

Generic tests are expected to run on any POSIX file system. Only the system calls common to all file systems, such as open, write, fsync, and fallocate, are used in generic tests. Each test case examines the behaviors of several system calls with various arguments. For example, test case 'generic/001' examines 5 system calls (open, write, unlink, rename, and stat) with various arguments by creating, copying, and renaming a tree of files (from 1 B to 1000 KB). Although, generic tests use only the generic VFS system calls, many test cases cannot be run: btrfs, ext4, and XFS cannot execute 45, 22, and 9 out of 185 generic tests, respectively. This result is obtained from our experiment discussed in Sect. 2.2.

Shared tests examine the features supported by many, but not all, file systems. For example, ext3 and ext4 provide three options for journaling so that users can trade off reliability against performance. On the other hand, journaling in XFS does not provide such options. As a result, 'shared/272', which tests the per-file data journaling, is applied only to ext3 and ext4.

Specific tests examine the features specific to btrfs, CIFS, ext4, UDF, and XFS. Xfstests provides 93, 1, 13, 3, and 194 test cases for btrfs, CIFS, ext4, UDF, and XFS, respectively. The test cases in this category highly depend on file system specific tools, kernel interfaces such as `/proc`, `/sys`, and ioctls, and the internal structure (e.g. special file flags) of each file system. An example of a specific test is that for a snapshot in btrfs. Test case 'btrfs/001' examines this feature by creating a snapshot, modifying the original file content, and confirming the snapshot is kept intact.

2.2 Code Coverage

To measure the code coverage of xfstests, we execute it on btrfs, ext4, and XFS with HDD drives. A virtual machine is used to measure the coverage to mitigate the problems caused by kernel panic or disk corruption. The virtual ma-

Table 1 Experimental setup.

Virtual Machine	QEMU/KVM 2.6.0
Kernel	Linux 4.0
Xfstests	master as of May 26, 2015 (commit 78bbab9)
#CPU	2
Memory	1 GiB
Size of TEST partition	10 GiB
Size of SCRATCH partition	10 GiB
mkfs options	Default
mount options	Default

chine for the test has two disks[†]: one for the TEST partition and the other for the SCRATCH partition. For the test cases on the TEST partition, the mkfs and mount options are the default (e.g. inode size = 256 and journal_ioprio = 3 in ext4. See the list of default values in [10]–[16].) For the SCRATCH partition, the test cases create and mount a file system themselves with the appropriate specified options. Table 1 summarizes the experimental setup.

Code coverage of file systems must be measured carefully to not include the code not executed by file system test cases. The setup procedures, such as booting an operating system and setting up the test cases, must be excluded from the measurements. For this purpose, we reset the coverage counter just before running a test case. A suite for the test cases (test cases, tools, and log files) is placed outside a file system for which the coverage is measured. If the test suite is placed on the tested file system, the code executed to access the test suite is measured as covered. To avoid this problem, the test suite was placed on NFS in our experiment.

Code coverage is measured in terms of line coverage. Line coverage records which statement is executed during the test. Line coverage has several limitations. For example, it cannot distinguish the sub-expressions composed of logical operators from the rest of the statement. If the statement is executed, the entire expression is considered executed even though some of the sub-expressions are not evaluated. Despite these limitations, line coverage is widely used in coverage measurements because of its acceptable overheads.

Figure 1 shows the overall coverage of xfstests on btrfs, ext4, and XFS. While XFS records 81.86% coverage, btrfs and ext4 record 67.98%, and 67.80% coverage, respectively, which are lower than XFS. In total 72.34% of the code of the three file systems is covered, as shown in Fig. 1.

Figure 1 also shows the breakdown of the code coverage of each file system component. We borrow the classification of the components used by Lu et al. [5]. Table 2 shows the classification. We add component *util* to Lu et al.'s classification and have ten components in total. *Balloc* and *extent* are the components for allocation of physical and logical blocks. *Dir*, *file*, and *inode* are responsible

[†]A small volume (10 GiB) is used in our experiment. No test case caused any problems due to this setting because file systems split their disk space into several small regions to reduce the cost of managing metadata.

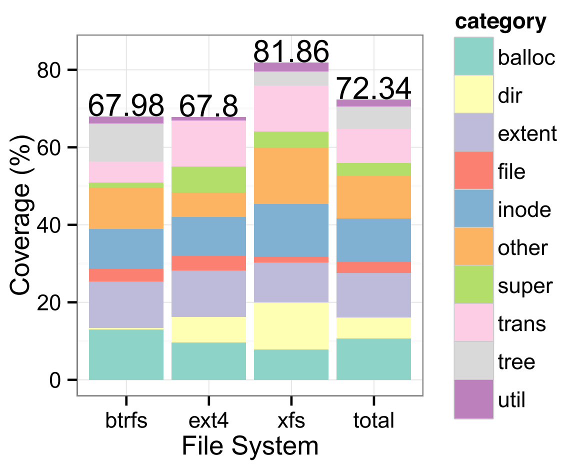


Fig. 1 Coverage overview.

Table 2 File system components.

balloc	Data block allocation and deallocation
extent	Contiguous mapping of physical blocks
dir	Directory management
file	File read and write operations
inode	Inode-related metadata management
trans	Journaling or other transactional support
super	Superblock-related metadata management
tree	Generic tree structures
util	Utility functions (hash, etc.)
other	Other file system features (e.g., xattr, ioctl, resize)

for basic file/directory operations. *Trans* manages file system transaction and its recovery. *Super* is for the super block management. *Tree* manages on-disk tree structures such as B-trees. *Util* contains utilities such as hash functions, and *Other* is the code to implement other features such as xattr or ioctl.

To understand which component contributes less or more to code coverage, we introduce the *contribution* of component k to the coverage. If all the components contribute to the coverage equally, the covered lines in k should be proportional to the total lines of that component. In other words, if a k occupies a percent of the entire code p, the covered code from k should occupy p percent of the entire covered code. If the percent of k in the covered code is higher than p, k contributes to the coverage more than the average. Otherwise, k contributes less than the average. The contribution of k, $Contrib(k)$, is defined as follows:

$$Contrib(k) = \frac{C_k}{\sum C_i} - \frac{L_k}{\sum L_i}$$

where C_k is the number of covered lines in k and L_k is the total number of lines in k. Summation moves all over the components. Suppose the code in k occupies 20% of all the code ($L_k/\sum L_i = 0.2$). If k occupies 80% of the covered code ($C_k/\sum C_i = 0.8$), then $Contrib(k) = C_k/\sum C_i - L_k/\sum L_i = 0.8 - 0.2 = 0.6$. If k occupies 10% of the covered code ($C_k/\sum C_i = 0.1$), then $Contrib(k) = C_k/\sum C_i - L_k/\sum L_i = 0.1 - 0.2 = -0.1$. If k gives higher coverage than the ratio of lines, *Contrib*(k) becomes positive. If not, *Contrib*(k) is negative.

Figure 2 shows the contribution of each component among the three file systems. All file systems commonly have negative contributions in *other* and *super*. Since *other* contains the code to implement features specific to a file system, low coverage is expected. *Super* contains the code to mount a file system. Since the test cases do not test the mount options extensively, the coverage of *super* is low. The details are discussed in Sect. 3.3.

Each file system shows different contributions in the other components. Btrfs achieves higher contribution in *tree*, *inode*, and *extent*. Ext4 achieves higher contribution in *extent*, *balloc*, *inode*, and *trans*. These components in btrfs and ext4 are core components for creating, opening, reading, and writing files. Thus, most of the test cases naturally execute these components.

XFS achieves high contribution in *inode*, *dir*, and *extent*, while achieving low contribution in *balloc*, *super*, and *other*. All the components achieving higher contributions are core components; however, *balloc*, which is considered a core component, is exceptional; its contribution is negative. This is due to the lack of coverage on the real-time device feature specific to XFS. This feature uses a special algorithm for block allocation; thus, the contribution of *balloc* is negative. It is notable that the variance of contributions is significantly lower in XFS (9.21×10^{-5}) than btrfs (7.40×10^{-4}) and ext4 (4.91×10^{-4}). This implies every component is tested in XFS better than in the other file systems because xfstests was originally developed for XFS.

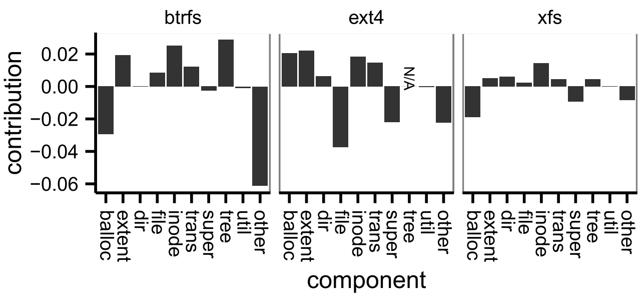


Fig. 2 Contribution of each component in btrfs, ext4, and XFS.

3. Analysis of Uncovered Code

This section investigates the code not covered by xfstests and unveils the reasons the code coverage is low. In xfstests, 12,400, 5,883, and 4,949 lines of code are not tested on btrfs, ext4, and XFS, respectively; 23,232 lines of code are not tested in total. To understand and classify the causes of low coverage, we manually investigate each line of the code not covered. Section 3.1 shows the classification of the causes, and Sects. 3.2 and 3.3 show the detailed analysis of the uncovered code.

3.1 Classification of Uncoverage Causes

After investigating each line of the code not covered by xfstests, we classified the causes of the uncoverage into seven categories: *MissingFeature*, *EnvDependency*, *Configuration*, *Evolution*, *OnDiskState*, *ErrorHandling*, *OutOfScope*, and *Unknown*. Table 3 shows the classification.

MissingFeature is the case in which test cases for some features are missing in the test suite. The code lines are not covered because xfstests does not have any tests to check certain features of file systems. *MissingFeature* can be further classified into two categories: *MissingFeature (common)* and *MissingFeature (specific)*. The test suite lacks test cases for common features in the former. Test cases for file-system-specific features are missing in the latter.

EnvDependency is the case in which a test case requires special support of hardware devices or special setting of the operating system. For example, some test cases require the TRIM command to be executed, which is supported only in SSDs; thus, cannot be executed on HDDs. Likewise, some test cases require special configurations of the operating system. For example, a test case for XFS checks the feature available only when SELinux is turned on. *EnvDependency* is classified into two categories: *Env-*

Table 3 Classification of uncoverage causes.

Category	Sub-category	Description
MissingFeature		Test cases for some features are missing
	common	Missing feature is common to file systems
	specific	Missing feature is specific to a file system
EnvDependency		Test case requires special support of hardware or special setting of operating system
	storage	Depends on configuration of storage devices
	os	Depends on configuration of operating system
Configuration		Depends on configuration of file systems (mkfs/mount)
Evolution		Data formats on disk have been updated
OnDiskState		Depends on specific states of on-disk structures
	allocation	Block allocation changes its execution paths depending on on-disk states
	recovery	Data recovery changes its execution paths depending on corrupted disk states
ErrorHandling		Error handling
OutOfScope		Excluded from coverage measurement
	dead	Dead code
	init	Initializing and cleaning up of file-system subsystem
	debug	Code used only for debugging

Dependency (storage) and *EnvDependency* (os). The former depends on the environment of physical storage devices. The latter depends on the configuration of the operating system.

A file system changes its behavior according to file system configurations. *Configuration* is the category for the code not covered due to the configuration — such as mkfs options and mount options — at the file system level. For example, if a “dirsync” option is turned on for the mount, all the updates on directories are performed synchronously. The execution paths change if “dirsync” is turned on; the `sync` function is called, which is not called if the option is off.

File systems evolve over time. The code not covered because of file system evolution is called *evolution*. A new version of a file system introduces, for instance, a new format of on-disk structures. To maintain backward compatibility, the new file system maintains the code for interpreting older versions of the on-disk format. If a disk is formatted with the new version, the code for interpreting the older versions is not executed during the test. In fact, btrfs has introduced a new format for metadata since version 3.10; thus, the code for the older version is not executed in xfstests.

OnDiskState is the code not covered because it is executed only under particular states of the on-disk data structures. A file system changes its execution paths based on the status of on-disk structures as well as in-memory status. For example, many file systems use on-disk B-trees to maintain metadata. A B-tree node is split when it becomes full. The code for splitting a node is not tested in most file operations because the node is seldom full.

ErrorHandling is the code for handling error cases. File systems handle I/O errors and memory allocation failures. The error handling code is not executed unless an error occurs. It is widely known that it is difficult to cover error handling code in test cases [17], [18]. While xfstests attempts to cover the error handling code by injecting faults, it is still far from covering the entire code.

OutOfScope is the code region for which it is impossible or meaningless to prepare test cases. For example, dead code remains in some file systems, which cannot be covered in test cases. Another example is the code for initializing the modules for file systems because they must be initialized before the tests are started. The code for debugging or tracing is also classified into *OutOfScope* in this paper.

3.2 Overall Comparison of Uncoverage Causes

Figure 3 plots the percentage of the uncoverage causes in each file system, and Table 4 shows the actual percentage of uncoverage causes. Interestingly, the dominant causes differ depending on the file system. For example, *EnvDependency* is the most major cause in btrfs, but it is minor in ext4. *EnvDependency* is major in btrfs because btrfs supports various disk layouts, such as file-system-level RAID to make use of multiple disks. The major cause in ext4 is *Configuration*. Ext4 provides many mkfs options such as MMP (Multi-Mount Protection) feature, metadata checksum, and inode inline data. In the test cases, all these options are not tested. The major cause in XFS is *ErrorHandling*. This suggests that XFS is tested better than other file systems because the error handling code is essentially difficult to prepare test cases.

Aside from the differences in the major causes, all file systems show similar tendency in *Evolution* and *ErrorHandling*. In all the tested file systems, *Evolution* is the least cause of the uncoverage. This is because it is uncommon to introduce new file formats. *ErrorHandling* is in the first or second position among all the file systems. This conforms to the fact that it is quite difficult to prepare the test cases for error handling [17], [18].

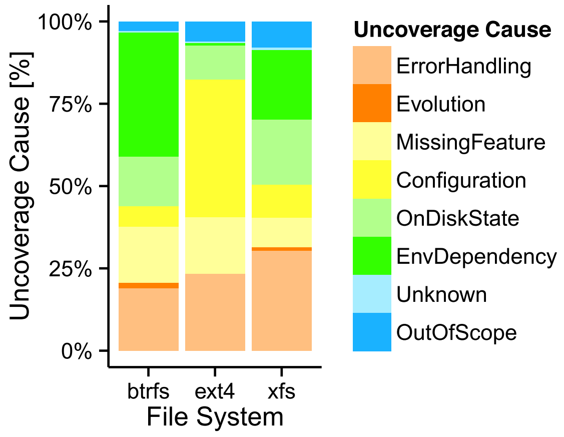


Fig. 3 Overall uncoverage measurement.

Table 4 Percentage of uncoverage causes.

btrfs		ext4		XFS	
EnvDependency	37.71%	Configuration	41.88%	ErrorHandling	30.43%
ErrorHandling	18.92%	ErrorHandling	23.32%	EnvDependency	21.14%
MissingFeature	16.90%	MissingFeature	17.10%	OnDiskState	19.76%
OnDiskState	15.19%	OnDiskState	10.35%	Configuration	10.04%
Configuration	6.17%	OutOfScope	6.00%	MissingFeature	8.99%
OutOfScope	2.81%	EnvDependency	0.70%	OutOfScope	7.84%
Evolution	1.82%	Unknown	0.54%	Evolution	1.01%
Unknown	0.49%	Evolution	0.10%	Unknown	0.79%

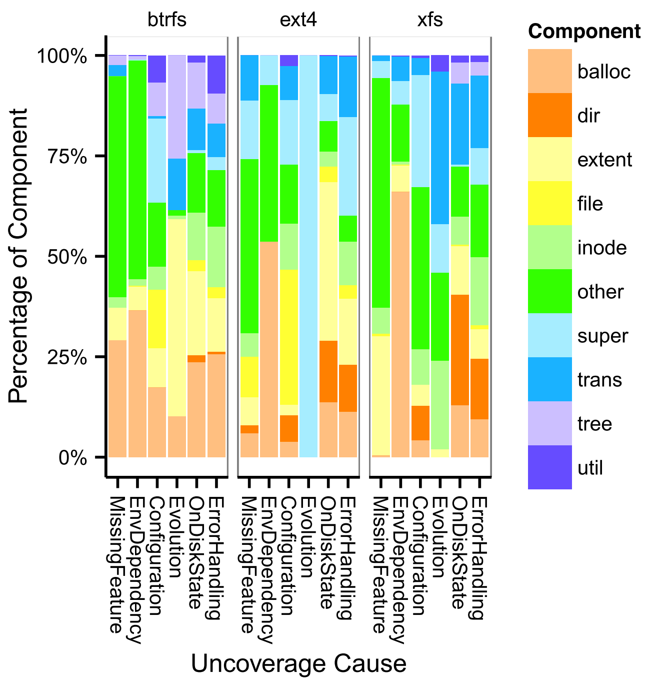


Fig. 4 Source of uncoverage.

3.3 Analysis of Uncovered Code

Finally, we investigate the details of the uncoverage causes. Figure 4 shows the distribution of uncovered lines among file system components for each uncoverage cause. For example, we can see from the figure that *Evolution* of ext4 is dominated by the uncovered lines of *super*.

3.3.1 *MissingFeature*

Some file system features are not tested in xfstests. To improve the coverage, it is necessary to add test cases to test the missing features. Note that there are some cases in which a certain feature is tested but all the options are not tested.

On btrfs, *MissingFeature* is mainly from *other* (54.94%) and *balloc* (29.12%). The code not covered in *other* is related to two features: 1) the seeding device, which generates a new btrfs file system from existing devices, and 2) the operations under '/sys/fs/btrfs'. This directory stores the configurations of btrfs; thus, writing files under the directing affects the behavior of btrfs. The test cases for those operations must be prepared.

The code in *balloc* is not covered because some options are not tested. For example, btrfs provides a command to balance block allocation among block groups. The test cases include those to test this command, but all the options are not tested. For example, the option to limit the number of block groups to be investigated is not tested.

On ext4, nearly half of *MissingFeature* is from *other* (43.44%) and *super* (14.51%). The code not covered in *other* implements the features used seldomly, and the test cases for them are missing. For example, ext4 allows a file to be hidden from the file system hierarchy. This feature is useful to hide and protect a boot loader from malicious users. This feature is not tested in xfstests. The code not covered in *super* is related to the operations to access the configurations under '/proc/fs/{ext4, jbd2}'. This is similar to the btrfs cases. In XFS, the code in *other* (57.08%) and *extent* (29.66%) contributes to lower coverage in *MissingFeature*. XFS allows users to attach extended attributes (name:value pairs) to files. The code not covered in *other* and *extent* are related to the extended attributes in XFS. On XFS, users can batch multiple requests to get/set/read the attributes. While the code to read the attributes is covered, the code to get or remove them is not covered. The code not covered in *extent* is also related to the file attributes. XFS takes an execution path different from the normal one if an attribute is attached to a large file. This code path is not covered although the path to attach an attribute to a small file is covered.

3.3.2 *EnvDependency*

EnvDependency is the category in which the code is not executed because special storage devices or operating system settings are necessary. Among all the investigated file systems, *other* and *balloc* are dominant in *EnvDependency*, though *other* on XFS is smaller than the other two file systems. *Other* takes 54.47%, 39.02%, and 14.24% of *EnvDependency*, respectively in btrfs, ext4, and XFS. *Balloc* takes 36.57%, 53.66%, and 66.06%, respectively.

Btrfs supports multi-device features such as RAID, Scrub, and device replacing. These features allow us to create a single file system over multiple devices without relying on other tools. These features are not tested in btrfs because we does not use multiple devices in the experiments. In ext4, the code for SSD setup is not executed in *balloc*. To execute the code, we have to prepare SSDs. The code not executed in *other* is for SELinux. To test the code, we have to turn on SELinux. XFS supports 'real-time' files. To use those files, a disk dedicated for real-time files must be prepared; thus, the code for real-time files are not executed in the experiment. Another cause is the DMAPI support for Hierarchical Storage Management. To enable it, we have to install the DMAPI kernel module in Linux. This is rarely used in Linux, and the kernel module is almost obsolete.

3.3.3 *Configuration*

File systems have quite a large configuration space; when a file system is created, there are many options to enable/disable various features. When a file system is mounted, many options should be specified. Depending on the configurations, file systems change their behaviors, and the code covered by the test cases varies. Our results indicate that a non-negligible part of the file system is not covered due to such mkfs or mount options. Xfstests encourages the testers to test mkfs/mount options on the TEST partition, but it is not easy to test all combinations of mkfs/mount options.

In particular, it is daunting to test all mkfs options because many file systems must be created with various options.

Among all the investigated file systems, *super* takes a large part in *configuration*. It takes 20.92%, 16.15%, and 27.97%, respectively in btrfs, ext4, and XFS. *Super* contains code to parse mount and mkfs options to verify the setup (e.g. to check if specified size of block is power of two) and enable/disable file system features. These lines of code in *super* count up in *Configuration*.

Aside from *super*, each file system has its own component to blame. In btrfs, *balloc* (17.39%) follows *super* (20.92%). Btrfs has an option to specify the beginning address of file-system blocks to protect an MBR region. When the option is specified, the calculation method of unallocated space is changed. In ext4, *file* (33.60%) is dominant because it contains experimental code to pack small file data into inode regions. A development version of a userland tool is required to enable the mkfs option. In XFS, *other* (40.24%) is dominant because the code for XFS-specific ioctls is not executed. XFS supports 32-bit and 64-bit versions of ioctls. Since our system runs a 64-bit environment, the code for the 32-bit version is not executed.

3.3.4 *Evolution*

All file systems we investigate support multiple metadata formats and the conversion between those formats. Btrfs has introduced a new type of metadata since Linux 3.10 [19]. It reduces the metadata size by 30–35% and is enabled by default. It implements the conversion code from old formats to the new one. This code is never executed unless the target file system is explicitly created with the old format. Nearly half of *evolution* comes from *extent* (49.12%), which handles the old metadata.

Ext4 supports the old scheme of the block mapping used in ext2 and ext3 for backward compatibility. It implements the conversion from the old scheme to the new one. Since the block mapping is converted at mount time, all the code for the conversion resides in *super*.

In XFS, *evolution* is spread into *trans* (38%), *inode* (22%), and *other* (22%). This comes from the fact that XFS has changed the format of the inode log in the journal. The code for journaling in XFS checks and converts the format of inode logs if necessary. Since *trans* is a module for journaling, the code not covered due to *evolution* resides in *trans* and *inode*.

3.3.5 *OnDiskState*

Even if the same test case is executed, a file system changes its execution paths depending on the on-disk structures. As shown in Fig. 4, all the file systems have major sources of *OnDiskState* in *balloc*, *dir*, and *extent*. These components are related to the allocation of disk space and called ‘allocation code’ altogether. Another source of *OnDiskState* is in *trans*, which is related to recovery.

Allocation Code. File systems use various data structures on disk, i.e., extents, trees, and bitmaps. These structures are allocated/freed on demand. In btrfs, *OnDiskState* related to the allocation spreads into *balloc* (23.63%) and *extent* (20.87%). Btrfs extensively uses B-trees to keep key/value pairs on disk. An execution path changes, for example, when a B-tree node becomes full. If a B-tree node is full when a new item is added, it is split into two nodes. This code for splitting a node is executed only when a B-tree node becomes full.

Ext4 has major sources of *OnDiskState* related to the allocation in *extent* (39.41%), *dir* (15.44%), and *balloc* (13.63%). An example of the execution path rarely taken is related to renaming a file. Ext4 manages directory entries with a B-tree whose key is a hash of a file name. When a file is renamed, the old entry is removed and the new one is added to the directory. If the B-tree node is full, it is split into two nodes and ext4 starts searching for an empty entry from the root of the B-tree. This code for searching from the root is not covered in our experiment because the B-tree node was not full.

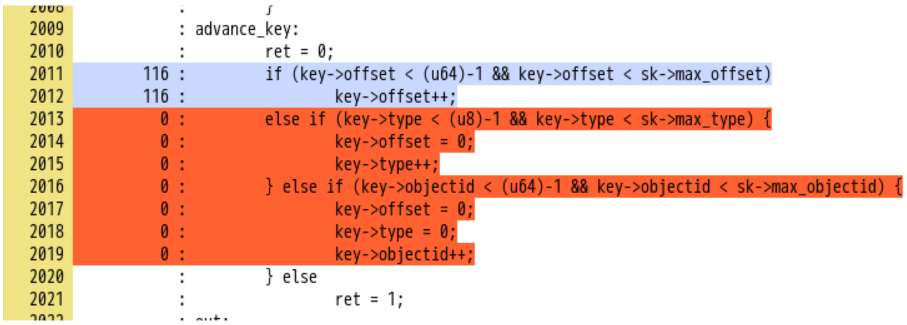
XFS has major sources of *OnDiskState* related to the allocation in *dir* (27.51%), *balloc* (12.99%), and *extent* (12.07%). XFS uses five different directory formats to trade off performance and disk utilization. It converts the format of a directory from one to another depending on the number of files in the directory. Unlink() system call involves the code to pack directory entries when the conversion of the directory formats occurs. In our experiment, unlink() system call is tested but the code for packing the directory entries is not executed because the code is executed only when the emptiness of the directory entries meets a specific condition.

Recovery Code. Recovery code in *trans* handles exceptional cases and is not tested well with test cases. *Trans* takes part of *OnDiskState* in 10.30%, 9.36%, and 20.25% in btrfs, ext4 and XFS, respectively.

A critical issue in testing the recovery code is that the recovery procedure diversifies because on-disk structures can be corrupted arbitrarily; thus, preparing test cases that can cover all possible processes of recovery is not easy. For example, btrfs manages each file with three different sets of metadata, and each set has different keys to be stored in a single B-tree. As a result, all sets of metadata belonging to the same file are placed either on a single disk block, on two blocks, or on three blocks. The recovery process of btrfs handles all cases of the metadata placements, but the test cases fail to attempt all possible cases. XFS has twice as many *OnDiskState* from *trans* as other file systems. This is because XFS uses logical journaling, which records the operations applied to file systems instead of the binary images of the updated disk blocks. While logical journaling reduces the amount of journaling regions, it complicates the recovery procedures. This leads to missing test cases in XFS.

3.3.6 *ErrorHandling*

ErrorHandling is the first or second major cause of uncover-



```
2009        : advance_key:
2010        :         ret = 0;
2011    116 :         if (key->offset < (u64)-1 && key->offset < sk->max_offset)
2012    116 :                 key->offset++;
2013      0 :         else if (key->type < (u8)-1 && key->type < sk->max_type) {
2014      0 :                 key->offset = 0;
2015      0 :                 key->type++;
2016      0 :         } else if (key->objectid < (u64)-1 && key->objectid < sk->max_objectid) {
2017      0 :                 key->offset = 0;
2018      0 :                 key->type = 0;
2019      0 :                 key->objectid++;
2020        :         } else
2021        :                 ret = 1;
```

Fig. 5 Coverage of btrfs B-tree searching ioctl. Blue lines are covered and red lines are not covered. Leftmost numbers are line numbers, and next column indicates its execution count.

age among all three file systems. It takes 18.92%, 23.32%, and 30.43% of uncoverage respectively in btrfs, ext4, and XFS. The code not covered due to *ErrorHandling* spreads over all the components. It is widely recognized that the error handling code is not easy to cover in test cases. The test suite for file systems is not exceptional; the error handling code is not covered well.

3.3.7 *OutOfScope*

There are interesting points in the code that are out of the scope of our investigation. All the file systems have dead code. Btrfs has dead code in *util*, *inode*, and *extent*. The dead code in *util* is that for debugging and the dead code in *inode* and *extent* is that for handling an option passed to functions. Since btrfs has no code that specifies certain options, the code handling those options is dead code. Ext4 has dead code in *trans*. It uses JBD2 for journaling, but JBD2 is designed to work for file systems other than ext4. The code not executed by ext4 becomes dead code in JBD2. XFS has the largest percentage of dead code (2.53%). It shares the code with userland tools. There are many functions not called from the kernel but called from userland tools. As a result, these functions become a large source of dead code.

3.3.8 Finding Bug

Our analysis of the uncovered code reveals a bug in btrfs. The bug is in the code that checks if a given key is in a specified range. In btrfs, the key is represented as a 136-bit integer that consists of a 64-bit object-id (more significant bits), 8-bit type, and 64-bit offsets (less significant bits). In the code shown in Fig. 5, there are three conditions at lines 2,011, 2,013 and 2,016. The intention with this code is to check if each part of the key (object-id, type, and offset) falls in a specified range; the first condition checks the offset (less significant bits) and the third condition checks the object-id (more significant bits). However, Fig. 5 indicates only the first condition is taken since covered are the blue lines and uncovered are red lines. This is strange because more significant bits (object-id) must be checked to decide if a given key is in a specified range, but the coverage report shows that only the less significant bits (offset) are checked.

This code is used for searching a B-tree for a key in a specified range. Btrfs continues the search even after the key goes beyond the specified range and results in a performance bug, which is generally difficult to find [20].

4. Discussion

Our analysis reveals five obstacles to improve the quality of test cases for file systems: 1) file-system specific features, 2) environmental dependency, 3) file-system configurations, 4) on-disk structures, and 5) file-system evolution. In this section, we discuss strategies for improving the quality of test cases. We believe our analysis results can be applied to file systems other than ext4, btrfs, and xfs because the obstacles we have identified are not peculiar to ext4, btrfs, and xfs.

First, our analysis reveals that the features specific to each file system are not tested well because of the lack of test cases. Adding more test cases for those features will be effective in improving test quality. The second obstacle is environmental dependency, in which a special storage device is needed to run some lines of code. To overcome this obstacle, using virtual machines that emulate a wide variety of storage devices is promising. The third obstacle is file-system configurations where some lines of code are not covered due to configurations. Since there is a huge number of valid combinations of configurations, it is not practical to run test cases on all possible combinations. A possible solution is to develop a code analyzer that discovers missing combinations from uncovered code. We speculate that the code analysis technique, which detects erroneous configurations [21], can be extended for this purpose. The fourth obstacle is that a code execution path depends on on-disk structures. To test file systems, we believe “test disks” should be prepared in addition to test cases; each test disk embodies a subtle corner case of on-disk structures. If a single test case is executed repeatedly for each test disk, it would cover the code that is executed only in a corner case. Virtual machines would be helpful in generating test disks. Finally, the on-disk format is sometimes changed for new features or better performance or reliability. We believe “test disks” are also useful to mitigate this obstacle; some test disks are formatted with an older disk format, which allows us to test the code for the older version.

File systems could be designed to help the development of test cases. For example, the error-handling code can be covered if the file system is equipped with fault injection. Fault injection is a technique to introduce synthetic faults. In file systems, disk I/O errors can be introduced by returning an error code intentionally from a device driver. A similar technique can be used to generate a subtle state of on-disk structures; the shortage of inode entries or disk spaces could be produced by rewriting the content of a disk block returned from a device driver.

5. Related Work

The key insight behind our paper is that understanding the causes of uncoverage helps to develop a better suite of test cases that can catch more bugs at the test phase. This section introduces other approaches to improve the quality of file systems. These approaches are complementary to our analysis of the uncovered code, and it is clarified how our results can be used to improve them or facilitate their use.

Modern file systems are too large and complex to be bug-free. Palix et al. [4] studied bug reports of the Linux 2.6 series and pointed out that file systems are one of the hotbeds of the kernel bugs. They identified that in the late versions of Linux 2.6 series, file systems have higher fault rate than drivers, which have the highest fault rate in the previous Linux kernels.

Lu et al. [5] investigated 5079 patches for Linux file systems (Ext3, Ext4, XFS, Btrfs, ReiserFS and JFS) of Linux 2.6 series. This investigation supports the fact that file systems are a hotbed of bugs by revealing nearly 40% of the patches are for bug fixes.

To improve the code quality of file systems, many researchers in the operating system community have developed many techniques to identify semantic bugs, which violate high-level rules or invariants peculiar to file systems. Some research projects targeted semantic bugs commonly observed in file systems. Chopper [22] focused on costly performance problems caused by poor layouts on disks. Recon [23] prevented buggy operations at runtime involved in journaling, and minimizes the corruption of journals. Our analysis of the uncoverage gives a clue to new domains of semantic bugs that should be addressed in the future.

SWIFT [24] is a testing tool for file system checkers, such as fsck, and automatically generates test cases to achieve higher coverage. To apply this technique to file systems, it must be extensively extended because file system checkers are concerned only with the recovery of corrupted file systems not the whole functionality of file systems.

Model checking is a promising approach to find corner-case bugs because it can verify that every possible path meets the invariants. Yang et al. [25], [26] applied model checking to existing file systems to find semantic bugs. SybylFS [27] detects buggy behaviors not allowed when a sequence of system calls is executed. To identify buggy behaviors, SybylFS defines a mathematical model of file system behavior and validates the trace of the execution of the sequence of system calls. SybylFS does not model special files, asynchronous I/O, or the corner cases such as resource exhaustion.

A well-known drawback of model-based approaches is that developers should manually provide the correct semantics of code for checking. Unfortunately, creating such semantics is not easy and is time-consuming and error-prone. Our analysis of uncovered code suggests code locations for which it is difficult to prepare test cases. It may be interesting to concentrate our efforts on building a model only for those code locations; the code that can be tested easily by test cases can be excluded from model checking.

JUXTA [28] is a tool that automatically infers high-level semantics from source code. The key insight behind JUXTA is that different implementations of the same functionality should obey the same system rules or semantics. By comparing existing implementations of Linux file systems, JUXTA derives their latent semantics and detects 118 bugs. This approach is useful to check common features of file systems but not helpful in finding bugs in file system specific features. Our analysis of the uncovered code shows file system specific features are not tested well, implying the need of finding bugs in such features.

To get rid of difficulties in verifying existing code base of file systems, novel file systems have been proposed whose design goes well with verification. BilbyFs [29] uses a highly modular design of a file system so that each module can be verified against high-level specification. FSCQ [6] is a file system with machine-checkable proof. COGENT [30] is a restricted language to write formally verifiable file-system code. Yggdrasil [31] is a toolkit to help programmers write verifiable code for file systems. Most of these file systems are implemented at user-level using FUSE [32]. This design simplifies the overall design of file systems compared with existing ones implemented at kernel level.

We believe new file systems or novel features will continue to be developed to meet the ever-changing demands on file systems. Modern desktop applications show characteristics different from traditional workloads on file systems [33]. Atlidakis et al. [34] pointed out modern operating systems are migrating to higher-level abstraction like SQLite [35]. Understanding the difficulties in testing file systems will help future programmers debug new features or novel file systems.

6. Conclusion

We measure the coverage of xfstests, a test suite for Linux file systems, on three file systems (btrfs, ext4, and XFS) and reveal that 23,232 lines of code are not covered. To understand why the coverage cannot be high, the uncovered lines of code are manually investigated. Our findings are three-fold:

- It is an overwhelming task to prepare test cases to cover all the features provided by file systems because each file system provides many file system specific features and some features can be tested on only special storage devices such as SSDs.

- Execution paths of file systems are highly dependent on mkfs and mount options. It is daunting to test all the options and their possible combinations because a new file system must be created and mounted for each combination. In addition, the execution paths depend on internal on-disk states, which cannot be controlled directly from the test cases.
- File systems sometimes update on-disk formats. To maintain backward-compatibility, file systems provide the code to access old formats. This code can be executed only when a file system encounters the old formats.

Our findings are useful to improve the coverage of test suites for file systems. Some issues can be addressed simply, for example, by adding new test cases but others cannot be addressed easily. We hope our result fosters the development of new methodologies for testing file systems. For example, we can develop a tool for model checking that focuses only on execution paths for which we cannot prepare test cases.

Acknowledgments

This work is partially supported by JSPS KAKENHI Grant Numbers JP16J03272, JP16K00104.

References

[1] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," Proceedings of Linux Symposium, pp.21–33, 2007.

[2] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," ACM Transactions on Storage, vol.9, no.3, pp.1–32, Aug. 2013.

[3] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," USENIX ATC, 1996.

[4] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall, "Faults in Linux 2.6," ACM Transactions on Computer Systems, vol.32, no.2, pp.1–40, June 2014.

[5] L. Lu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and S. Lu, "A Study of Linux File System Evolution," ACM Transactions on Storage, vol.10, no.1, pp.1–32, Jan. 2014.

[6] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M.F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system," Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, pp.18–37, ACM, 2015.

[7] "[patch] ext4: Fix data corruption caused by unwritten and delayed extents." http://article.gmane.org/gmane.linux.kernel.stable/131959.

[8] sgi, "oss.sgi.com Git - xfs/cmds/xfstests.git/summary." http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git;a=summary, 2014.

[9] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, New York, NY, USA, pp.72–82, ACM, 2014.

[10] "Manpage/mkfs.btrfs - btrfs Wiki." https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs, 2018.

[11] "e2fsprogs/mke2fs.conf.in at master tytso/e2fsprogs." https://github.com/tytso/e2fsprogs/blob/master/misc/mke2fs.conf.in, 2018.

[12] "mke2fs(8) - Linux manual page." http://man7.org/linux/man-pages/man8/mke2fs.8.html, 2018.

[13] "mkfs.xfs(8) - Linux manual page." http://man7.org/linux/man-pages/man8/mkfs.xfs.8.html, 2018.

[14] "Manpage/btrfs(5) - btrfs Wiki." https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs(5)#MOUNT_OPTIONS, 2018.

[15] "ext4(5) - Linux manual page." http://man7.org/linux/man-pages/man5/ext4.5.html, 2018.

[16] "xfs(5) - Linux manual page." http://man7.org/linux/man-pages/man5/xfs.5.html, 2018.

[17] J.J. Bai, Y.P. Wang, J. Yin, and S.M. Hu, "Testing error handling code in device drivers using characteristic fault injection," 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, pp.635–647, USENIX Association, 2016.

[18] M.M. Swift, B.N. Bershad, and H.M. Levy, "Improving the reliability of commodity operating systems," SIGOPS Oper. Syst. Rev., vol.37, no.5, pp.207–222, Oct. 2003.

[19] "Btrfs: add a incompatible format change for smaller metadata extent refs." https://lwn.net/Articles/542432/.

[20] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, New York, NY, USA, pp.77–88, ACM, 2012.

[21] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, New York, NY, USA, pp.687–700, ACM, 2014.

[22] J. He, D. Nguyen, A.C. Arpaci-dusseau, R.H. Arpaci-dusseau, W. Madison, and S. Clara, "Reducing File System Tail Latencies with Chopper," Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST'15), 2015.

[23] D. Fryer, K. San, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A.D. Brown, "Recon: Verifying File System Consistency at Runtime," Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12), pp.7–20, 2012.

[24] J.C.M. Carreira, R. Rodrigues, G. Candea, and R. Majumdar, "Scalable testing of file system checkers," Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12, pp.239–252, ACM Press, April 2012.

[25] J. Yang, C. Sar, and D. Engler, "Explode: A lightweight, general system for finding serious storage system errors," Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, pp.131–146, USENIX Association, 2006.

[26] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," ACM Transactions on Computer Systems, vol.24, no.4, pp.393–423, Nov. 2006.

[27] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell, "SibylFS," Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15, pp.38–53, ACM Press, Oct. 2015.

[28] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking Semantic Correctness: The Case of Finding File System Bugs," Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15, pp.361–377, 2015.

[29] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser, "File systems deserve verification too!," Proceedings of the Seventh Workshop on Programming Languages and Operating Systems - PLOS '13, pp.1–7, 2013.

[30] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser, "Cogent: Verifying high-assurance file system implementations," Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, New York, NY,

USA, pp.175–188, ACM, 2016.

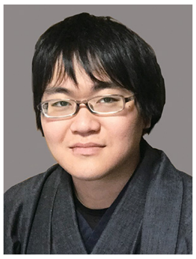
[31] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-button verification of file systems via crash refinement," 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), GA, pp.1–16, USENIX Association, 2016.

[32] "Fuse. filesystem in userspace." https://github.com/libfuse/libfuse.

[33] T. Harter, C. Dragga, M. Vaughn, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications," ACM Transactions on Computer Systems, vol.30, no.3, pp.1–39, 2012.

[34] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "POSIX abstractions in modern operating systems: The Old, the New, and the Missing," Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16, New York, New York, USA, pp.1–17, ACM Press, April 2016.

[35] "SQLite." https://www.sqlite.org/.



Naohiro Aota received his B.E. degree from the Department of Information and Computer Sciences at Osaka University in 2014, and M.E. degree from the Department of Information and Computer Science at Keio University in 2016. He is currently a Ph.D student at the School of Science for Open and Environmental Systems at Keio University. His research interest includes file systems, operating systems and storage systems.

Kenji Kono received his BSc degree in 1993, MSc degree in 1995, and PhD degree in 2000, all in computer science from the University of Tokyo. He is a professor in the Department of Information and Computer Science at Keio University. He received the IPSJ Yamashita-Memorial Award in 2000, IPSJ Annual Best Paper Awards in 1999, 2008, 2009, and 2012, JSSST Software Paper Award in 2014, IBM Faculty Award in 2015, and JSSST Basic Research Award in 2016. He served as a PC member of top conferences such as ICDCS and DSN. He also organized ACM SIGOPS APSys in 2015. His research interests include operating systems, system software, and computer security. He is a member of the IEEE, ACM and USENIX.