Recon: Verifying File System Consistency at Runtime

Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng,
Shaun Benjamin, Ashvin Goel, Angela Demke Brown
University of Toronto

Abstract

File system bugs that corrupt file system metadata on disk are insidious. Existing file-system reliability methods, such as checksums, redundancy, or transactional updates, merely ensure that the corruption is reliably preserved. The typical workarounds, based on using backups or repairing the file system, are painfully slow. Worse, the recovery is performed long after the original error occurred and thus may result in further corruption and data loss.

We present a system called Recon that protects file system metadata from buggy file system operations. Our approach leverages modern file systems that provide crash consistency using transactional updates. We define declarative statements called consistency invariants for a file system. These invariants must be satisfied by each transaction being committed to disk to preserve file system integrity. Recon checks these invariants at commit, thereby minimizing the damage caused by buggy file systems.

The major challenges to this approach are specifying invariants and interpreting file system behavior correctly without relying on the file system code. Recon provides a framework for file-system specific metadata interpretation and invariant checking. We show the feasibility of interpreting metadata and writing consistency invariants for the Linux ext3 file system using this framework. Recon can detect random as well as targeted file-system corruption at runtime as effectively as the offline e2fsck file-system checker, with low overhead.

1 Introduction

It is no surprise that file systems have bugs [20, 29, 31]. Modern file systems are designed to support a range of environments, from smart phones to high-end servers, while delivering high performance. Further, they must handle a large number of failure conditions while preserving data integrity. Ironically, the resulting complexity leads to bugs that can be hard to detect even under heavy testing. These bugs can cause silent data corruption [20, 19], random application crashes, or even worse, security exploits [30].

Unlike hardware errors and crash failures, it is much harder to recover from data corruption caused by bugs in file-system code. Hardware errors can be handled by using checksums and redundancy for error detection and recovery [4, 10]. Crash failure recovery can be performed using transactional methods, such as journaling [12], shadow paging [14], and soft updates [9]. Modern file systems, such as ZFS, are carefully designed to handle a wide range of disk faults [32]. However, the machinery used for protecting against disk corruption (e.g., checksums, replication and transactional updates) does not help if the file system code itself is the source of an error, in which case these mechanisms only serve to faithfully preserve the incorrect state.

File system bugs that cause severe metadata corruption appear regularly. We compiled a list of bugs in the Linux ext3 and the recently deployed btrfs file systems, by searching for "ext3 corruption" and "btrfs corruption" in various distribution-specific bug trackers or mailing lists. Based on the bug description and discussions, we removed bugs that did not cause metadata inconsistency, or were not reproducible, or were reported by a single user only. Table 1 summarizes the remaining bugs. Note that ext3, despite its maturity and widespread use, shows continuing reports of corruption bugs. One recent example is not yet closed, while another closed only in 2010 and affected the ext2, ext3 and ext4 file systems. These reports likely under-represent the problem because the bugs that cause metadata corruption may be *fail silent*, i.e., the error is not reported at the time of the original corruption. By the time the inconsistencies appear, the damage may have escalated, making it harder to pinpoint the problem.

When metadata corruption is discovered, it requires complex recovery procedures. Current solutions fall in two categories, both of which are unsatisfactory. One approach is to use disaster recovery methods, such as a backup or a snapshot, but these can cause significant downtime and loss of recent data. Another option is to use an offline consistency check tool (e.g., e2fsck) for restoring file system consistency. While a consistency check can detect most failures, it requires the entire disk to be checked, causing significant downtime for large file systems. This problem is getting worse because disk capacities are growing faster than disk bandwidth and seek time [13]. Furthermore, the consistency check is run after the fact, often after a system crash occurs or even less frequently with journaling file systems. Thus an error may propagate and cause significant damage, making repair a non-trivial process [11]. For example, Section 5 shows that a single byte corruption may cause repair to fail.

To minimize the need for offline recovery methods, our aim is to verify file-system metadata consistency at runtime. Metadata is more vulnerable to corruption by file

FS	Source	Bug Title	Closed
ext3	http://lwn.net/Articles/2663/	ext3 corruption fix	2002-06
ext3	kerneltrap.org/node/515	Linux: Data corrupting ext3 bug in 2.4.20	2002-12
ext3	Redhat, #311301	panic/ext3 fs corruption with RHEL4-U6-re20070927.0	2007-11
ext3	https://lkml.org/lkml/2008/12/6/88	Re: [2.6.27] filesystem (ext3) corruption (access beyond end)	2008-06
ext3	Debian, #425534	linux-2.6: ext3 filesystem corruption	2008-09
ext3	Debian, #533616	linux-image-2.6.29-2-amd64: occasional ext3 filesystem corruption	2009-06
ext3	Redhat, #515529	ENOSPC during fsstress leads to filesystem corruption on ext2, ext3, and ext4	2010-03
ext3	https://lkml.org/lkml/2011/6/16/99	ext3: Fix fs corruption when make_indexed_dir() fails	2011-06
ext3	Redhat, #658391	Data corruption: resume from hibernate always ends up with EXT3 fs errors	Not yet
btrfs	https://lkml.org/lkml/2009/8/21/45	btrfs rb corruption fix	2009-08
btrfs	https://lkml.org/lkml/2010/2/25/376	[2.6.33 regression] btrfs mount causes memory corruption	2010-02
btrfs	https://lkml.org/lkml/2010/11/8/248	DM-CRYPT: Scale to multiple CPUs v3 on 2.6.37-rc* ?	2010-09
btrfs	https://lkml.org/lkml/2011/2/9/172	[PATCH] btrfs: prevent heap corruption in btrfs_ioctl_space_info()	2011-02
btrfs	https://lkml.org/lkml/2011/4/26/304	btrfs updates (slab corruption in btrfs fitrim support)	2011-04

Table 1: File system bugs causing data corruption. All Red Hat and Debian bugs are rated high-severity. The severity level of bugs obtained from mailing lists is not known.

system bugs because the file system directly manipulates the contents of metadata blocks. Metadata corruption may also result in significant loss of user data because a file system operating on incorrect metadata may overwrite existing data or render it inaccessible.

We present a system called Recon that aims to preserve metadata consistency in the face of *arbitrary* file-system bugs. Our approach leverages modern file systems that provide crash consistency using transactional methods, such as journaling [28, 6, 27] and shadow paging file systems [14, 4, 16]. Recon checks that each transaction being committed to disk preserves metadata consistency. We derive the checks, which we call consistency invariants, from the consistency rules used by the offline file system checker. A key challenge is to correctly interpret file system behavior without relying on the file system code. Recon provides a block-layer framework for interpreting file system metadata and invariant checking.

An important benefit of Recon is its ability to convert fail-silent errors into detectable invariant violations, raising the possibility of combining Recon with file system recovery techniques such as Membrane [26], which are unable to handle silent failures.

Our current implementation of Recon shows the feasibility of interpreting metadata and writing consistency invariants for the widely used Linux ext3 file system. Recon checks ext3 invariants corresponding to most of the consistency properties checked by the e2fsck offline check program. It detects random and type-specific file-system corruption as effectively as e2fsck, with low memory and performance overhead. At the same time, our approach does not suffer from the limitations of offline checking described earlier because corruption is detected immediately. The rest of the paper describes our approach in detail and presents the results of our initial evaluation.

2 Approach

The Recon system interposes between the file system and the storage device at the block layer and checks a set of consistency invariants before permitting metadata writes to reach the disk. We derive the invariants from the rules used by the file system checker. As an example, the e2fsck program checks that file system blocks are not doubly allocated. Our invariants check this property at runtime and thus prevent file-system bugs from causing any double allocation corruption on disk.

Figure 1 shows the architecture of the Recon system. Recon provides a framework for caching metadata blocks and an API for checking file-system specific invariants using its metadata cache. A separate cache is maintained because the file system cache is untrusted and because it allows checking the invariants efficiently. Besides ext3, we have also examined the consistency properties of the Linux btrfs file system and implemented several btrfs invariants. The paper describes our initial experience with adapting our system for btrfs.

Our approach addresses three challenges: 1) *when* should the consistency properties be checked, 2) *what* properties should be checked, and 3) *how* should they be checked. Below, we describe these challenges and how we address them. The caching framework and the file-system specific Recon APIs are described in Section 4.

2.1 When to Check Consistency?

The in-memory copies of metadata may be temporarily inconsistent during file system operation and so it is not easy to check consistency properties at arbitrary times. Instead, checks can be performed when the file system itself claims that metadata is consistent. For example, journaling and shadow-paging file systems are already designed

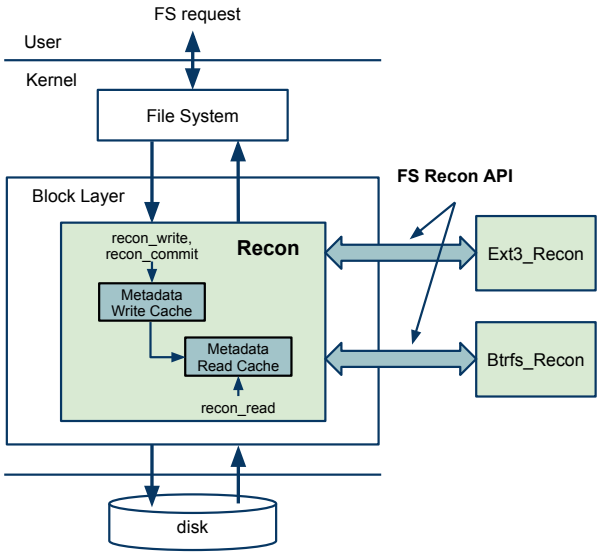

Figure 1: The Recon Architecture

to ensure crash consistency using transactional methods, wherein disk blocks from one or more operations, such as the creation of a directory and a file write, are grouped into transactions. Transaction commits are well-defined points at which the file system believes that it is consistent, and hence transaction boundaries serve as convenient vantage points for verifying consistency properties. Recon checks transactions *before* they commit, thereby ensuring that a committed transaction is consistent, even in the face of arbitrary file system bugs.[1]

Checking consistency for shadow paging systems is relatively straightforward because all transaction data is written to disk before the commit block. For example, btrfs writes all blocks in a transaction, and then commits the transaction by writing its superblock. Recon checks each transaction before the superblock is written to disk.

Checking consistency for journaling file systems is more complicated because transaction data is written to disk both before and after the commit block. For example, ext3 writes metadata to disk in several steps: 1) write metadata to journal, 2) write commit block to journal, at which point the transaction is committed, 3) write (or checkpoint) metadata to its final destination on disk, and 4) free space in the journal.

During Step 1, Recon copies metadata blocks into its write cache, giving it a view of all the updates in a transaction. Then it checks the ext3 transaction in Step 2, i.e., before the commit block is written to the journal, which ensures that all blocks in the transaction are checked for consistency before they become durable. Checking consistency after commit could lead to checkpointing a corrupt block, and furthermore it would not be possible to undo such corruption. Besides checking consistency at commit, we also need to verify the checkpointing process. This step requires checking that all the committed blocks and their contents are checkpointed correctly.

[1] Implementing consistency invariants for soft update file systems [9] that provide consistency after each write but do not use transactions should be possible but will likely be more complicated.

2.2 What Consistency Properties to Check?

Identifying the correct consistency properties is challenging because the behavior of the file system is not formally specified. Fortunately, we can derive an informal specification of metadata consistency properties from offline file-system consistency checkers, such as the Linux e2fsck program. For example, Gunawi et al. found that the Linux e2fsck program checks 121 properties that are common to both ext2 and ext3 file systems and some ext3 journal properties and optional features [11].

These consistency properties define what it means to have consistent metadata on disk. Our aim is to ensure that any metadata committed to disk will maintain these same consistency properties. Unfortunately, consistency properties are *global* statements about the file system. For example, a simple check implemented by e2fsck is that the deletion times of *all* used inodes are zero. Determining the in-use status of all inodes, and checking the deletion time of all used inodes is infeasible at every transaction commit. Similarly, another consistency property is that *all* live data blocks are marked in the block bitmap. Checking these global properties requires a full disk scan.

Instead, we derive a *consistency invariant* from each consistency property. The invariant is a local assertion that must hold for a transaction to preserve the corresponding file system consistency property. For example, consider the "all live data blocks are marked in the block bitmap" property. The corresponding consistency invariant is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block bitmap within the same transaction, i.e., the invariant is "block pointer set from 0 to N $\Leftrightarrow$ bit N set in bitmap". This invariant can be checked by examining only the updated blocks, i.e., the updated pointer block and the updated block bitmap must be part of the same transaction. We describe this invariant in more detail in Section 3.2.

We structure a consistency invariant as an implication, $A \Rightarrow B$. The premise A *always* involves an update to some data structure field, and hence checking the invariant is triggered by a change in that field. When such an update occurs then the conclusion B must be true to preserve the invariant. If a converse $B \Rightarrow A$ invariant also exists, then we refer to the two invariants as a biconditional invariant $A \Leftrightarrow B$, as shown in the example above.

We rely on the ability to convert consistency properties requiring global information into invariants that can be checked using information "local" to the transaction,

as described in the previous example. Such a transformation must be possible because file systems keep themselves consistent without examining the entire disk state. In other words, our invariant checking should not require much more data than the file system itself needs for its operations. Section 5 shows that this is indeed the case because Recon has low overheads.

Finally, our invariant checking approach relies on an inductive argument. It assumes that the file system is consistent before each transaction. If the updates in the transaction meet the consistency invariants, the file system will remain consistent after the transaction. Likewise, if an invariant is violated, there is potential for data loss or incorrect data being returned to applications. Section 2.4 provides more details about our assumptions.

2.3 How to Check Consistency Invariants?

Consistency invariants are expressed in terms of logical file-system data structures, such as current and updated values of block pointers, bits in block bitmap, etc.. However, Recon needs to observe physical blocks below the file system because it cannot trust a buggy file system to provide the correct logical data structure information. We bridge this semantic gap by inferring the types of metadata blocks when they are read or written, which allows parsing and interpreting them, similar to semantically smart disks [24]. Then Recon checks invariants on the typed blocks at commit points, as described below.

Metadata interpretation Block typing and metadata interpretation depend on the idea that file systems access metadata by following a graph of pointers. For example, a pointer to a block is read before the pointed-to block is read, which we call the pointer-before-block assumption. These pointers may be explicit block pointers or are implied by the structure of the file system. For example, ext3 will read an inode containing a pointer to an indirect block before reading the indirect block. When an inode block is read, Recon copies it into its read cache and then parses the inodes in the block to create a mapping from a block to its type for any metadata blocks pointed to by the inodes. In this case, Recon creates a block-type mapping associating the "indirect block" type with the block pointed to by the EXT3_IND_BLOCK pointer in the inode. As a result, Recon recognizes an indirect block when it is read.

Similarly, the block group descriptor (BGD) tables in ext3 describe the locations of inode blocks and inode and block allocation bitmaps. The BGD tables must be read before any of the blocks that they point to, allowing Recon to create block-type mappings for inode and bitmap blocks. This block-type identification is bootstrapped using the superblock, which exists at a known location.

When a metadata block is newly allocated in a transaction, Recon does not yet know its type. In this case, there must exist an updated metadata block in the transaction with a known type that points to this unclassified block directly or indirectly, or else the newly allocated block would not be reachable in the file system. By following the path of pointers from the known metadata block to the newly allocated block, Recon can always create block-type mappings for newly allocated blocks.

For example, suppose a block is allocated to an indirect block of a file. If the file already existed then its inode block must have been read and updated in the transaction. Since the inode block was read previously, Recon knows its type and can determine the type of the newly allocated indirect block. Similarly, if the file did not exist then its parent directory must have existed and been updated, which helps determine the types of the (possibly newly allocated) inode block and then the indirect block. Determining the types of newly allocated blocks may require multiple passes over the blocks updated in the transaction. At the end, all new metadata blocks must be typed or else the pointer-before-block assumption is violated.

Commit processing At commit, Recon uses the block-type mapping to determine the data structures in each of the (updated) transaction blocks, available in the Recon write cache. These data structures are compared with their previous versions, which are derived from the Recon read cache, at the granularity of data structure fields. Each field update generates a logical *change record* with the format [*type, id, field, oldval, newval*].

The *type* specifies a data structure (e.g., inode, directory block). The *id* is the unique identifier of a specific object of the given type (e.g. inode number). The (type, id) pair allows locating the specific data structure in the file system image. The *field* is a field in the structure (e.g. inode size field) or a key from a set (e.g. directory entry name). The *oldval* and *newval* are the old and new values of the corresponding field. These records are generated for existing, newly allocated and deallocated metadata blocks. When an item is newly created or allocated, the oldval is ϕ (a sentinel value). Similarly, when an item is destroyed or deallocated, the newval is ϕ.

Figure 2 shows an example of a set of change records associated with an ext3 transaction in which a single write operation increases the size of a file from one block to two blocks. Change records serve as an abstraction, cleanly separating the interpretation of physical metadata blocks from invariant checking on logical data structures. We show how invariants are implemented using change records in Section 3. When all invariants are checked successfully, the transaction is allowed to commit.

2.4 Fault Model

Our goal is to preserve file-system metadata consistency in the presence of arbitrary file-system bugs. We make

```
[Inode, 12, block[1], 0, 22717]      ; In inode 12, direct block ptr 1 is set to block 22717
[BBM, 22717, 0, 0, 1]                ; Block 22717 is marked allocated in block bitmap
[BGD, 0, free_blocks, 1500, 1499]    ; In block group 0, nr. of free blocks decreases by 1
[Inode, 12, i_size, 4010, 7249]      ; i_size field increases from 4010 to 7249 bytes
[Inode, 12, i_blocks, 8, 16]         ; i_blocks is the number of sectors used by file
[Inode, 12, mtime, 1-18-12, 1-20-12] ; timestamp change
[Inode, 12, ctime, 1-16-12, 1-20-12] ; timestamp change
```

Figure 2: Change records when a block is added to a file

three assumptions to provide this guarantee. First, we assume that the Recon code and its invariant checks are correct and immutable and the Recon metadata cache is protected. If these assumptions are incorrect, it is unlikely that an inconsistent transaction would pass our checks, because the file-system bug and our corrupted check would need to be correlated. However, Recon may generate false alarms, indicating corruption even when a transaction is consistent. Such corruption is still an indication of a bug in the overall system. A hypervisor-based Recon implementation would provide stronger isolation of the Recon code and data from the kernel, helping ensure metadata consistency in the face of *arbitrary kernel* bugs.

Second, if the ext3 file system writes a metadata block before Recon knows its type then Recon will assume that a data block is being written and will allow the operation. For example, a file system bug may corrupt the block number in a disk request structure and cause a misdirected write to a metadata block. Recon will not detect this error because the write violates our pointer-before-block assumption, and ext3 does not provide any other way to identify the block being updated.[2] As future work, we plan to retrofit ext3 to allow such identification. Misdirected writes will not cause a problem with btrfs because its extents are self-identifying [2].

Finally, our inductive assumption about metadata consistency before each transaction (discussed in Section 2.2) requires correct functioning of the lower layers of the system, including the Linux block device layer and all hardware in the data path. It is possible to detect and recover from errors at these layers by using metadata checksums and redundancy. This functionality could be implemented at the block layer for the ext3 file system [10]. The btrfs file system already provides such functionality [16]. If these assumptions are not met, offline checking and repair should be used as a last resort.

3 Consistency Invariants

A file system checker verifies file system consistency by applying a comprehensive set of rules for detecting and optionally repairing inconsistencies. We are primarily interested in checking consistency properties and can reuse the rules associated with detecting, but not repairing, inconsistencies. We have applied our approach to the ext3 and the btrfs file systems. Below, we provide an overview of the consistency rules for these file systems.

The SQCK system [11] encapsulates the 121 checks of the ext3 fsck program in a set of SQL queries. Although there is a close correspondence between SQCK queries and e2fsck checks, some SQCK queries combine multiple checks. Table 2 provides a breakdown of the number of rules checked by SQCK for different file-system data structures. We show 101 rules in Table 2, because the rest are used for repair. The simplest checks (lines starting with the word *Within*) examine individual structures (e.g., superblock fields, inode fields, and directory entries appear valid). Some checks ensure that pointers lie within an expected range. More complicated checks (lines starting with the word *Between*) ensure that block pointers (across all files) do not point to the same data blocks, and directories form a connected tree.

We have done a similar classification of the rules checked by the btrfs checker, as shown in Table 3. Btrfs is an extent-based, B-tree file system that stores file-system metadata structures (e.g., inodes, directories, etc.) in B-tree leaves [16]. It uses a shadow-paging transaction model for updates and for supporting file-system snapshots. Extent allocation information is maintained in an extent B-tree, which serves the same purpose as ext3 block bitmaps. The roots for all the B-trees are maintained in a top-level B-tree called the root tree. Although the btrfs checker is still a work in progress (e.g., it performs no repair), currently it uses 30 rules for detecting inconsistencies. Of these, the first four rule sets are used to check the structure of the B-tree, while the rest deal with typical file-system objects such as inodes and directories.

Next, we provide several examples that show how we transform the consistency properties for various data structures shown in Tables 2 and 3 into invariants. An invariant is implemented by pattern matching change records. When such a match occurs, some invariants accumulate bookkeeping information then require some final processing at transaction commit.

[2] We did not observe this problem because our fault injector corrupts metadata blocks but does not cause misdirected writes (see Section 5.2).

Datatype		#rules
A	Within superblock	23
B	Within block group descriptors (BGD)	5
C	Within a single inode	28
D	Within a single directory	14
E	Between inode and directory entries	5
F	Between inode and its block pointers	2
G	Between inode, inode bitmap, orphan list	3
H	Between block bitmap and block pointers	5
I	Between block, inode bitmap, BGD table	3
J	Between directories	4
K	Bad blocks inode	7
L	Extended attributes ACL	2

Table 2: Number of Ext3/SQCK rules by datatype

Datatype		#rules
A	Within tree block	2
B	Between parent and child tree blocks	3
C	Between extent tree and extents	3
D	Within an extent item in extent tree	2
E	Between inodes and file system trees	2
F	Between inode and directory entries	4
G	Between inodes, inode refs and dir. entries	2
H	Within directory entries	1
I	Between inode, data extents, checksum tree	6
J	Between inode and orphan items	1
K	Between root tree and file system trees	3
L	Between root tree and orphan items	1

Table 3: Number of Btrfs rules by datatype

3.1 Ext3 Immutable Fields, Range Checks

The ext3 fsck program checks for valid values in several fields of the superblock and group descriptor table (rows A and B in Table 2). Many of these fields are initialized when a file system is created and should never be modified by a running file system. Invariants on these fields are implemented by pattern matching a change record of the form [Superblock, _, immutable_field, _, _], where immutable_field is the name of the field that should not change, and _ matches any value. The existence of this record indicates that the field was modified, and signals a violation. Another similar class of consistency properties requires simple range checks on the values of given fields.

3.2 Ext3 Block Bitmap and Block Pointers

An important consistency properties in ext3 is that no data block may be doubly allocated, i.e., every block pointer (whether it is found in a live inode or indirect block) must be unique or 0. Checking this property would be expensive if we simply scanned all inodes and indirect blocks searching for another instance of the pointer.

The file system maintains this property without examining the entire disk state by using block allocation bitmaps (row H in Table 2), with the resulting consistency property being that "all live data blocks are marked in the block bitmap". The corresponding consistency invariant is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block bitmap within the same transaction, as shown below.

block pointer set to N from 0 $\Leftrightarrow$ bit N set in bitmap (1)
block pointer set to 0 from N $\Leftrightarrow$ bit N unset in bitmap (2)

These invariants involve relationships between different fields and require matching multiple change records. The left side of the first invariant is triggered by matching change records of the form [_, _ , block_pointer_field, 0, X], indicating a new pointer to block X. When such a match occurs, we insert a "new pointer" flag with key X into a rule-specific table. The right side of this (biconditional) invariant is triggered by matching [BBM, Y, _, 0, 1] records, indicating bit Y in the allocation bitmap is newly set. When this match occurs, we insert a "bit set" flag with key Y into the same table. During final processing, the implementation verifies that for each key in the table, both flags are set. Otherwise the invariant has been violated. For example, in the simple transaction shown in Figure 2, there is exactly one record matching each of the left and right sides of Invariant 1 shown above, and the values of X and Y are both 22717.

Invariants 1 and 2 ensure that when a block pointer is set, the corresponding bit in the bitmap is also set. However, we must also ensure that a pointer to the same block is set only once in a transaction, i.e., we must check for double allocation within a transaction. To do so, we simply count the number of times we see a block pointer set to a given block in the transaction:

block pointer set to N $\Rightarrow$
(count(block pointer==N) in transaction)==1 (3)

3.3 Ext3 Directories

The inter-directory consistency properties essentially ensure that the directory tree forms a single, bidirected[3] tree (row J in Table 2). This complex consistency property requires two biconditional and two regular invariants. Whenever a directory is linked (or its ".." entry changes), Invariant 4 checks that the directory's parent (child) has the directory as its child (parent). This check also ensures that a directory does not have multiple parents. When a directory is unlinked (or moved), Invariant 5 checks that it is unlinked on both sides (although not shown, we also check that an unlinked directory is empty). When a directory's "." entry is updated, Invariant 6 checks that the "." entry points to itself.

[Dir, C, "..", _, P] $\Leftrightarrow$ [Dir, P, nm, _, C] and (nm != "..") (4)

[3] A bidirected tree is the directed graph obtained from an undirected tree by replacing each edge by two directed edges in opposite directions.

[Dir, C, "..", P, _] ⇔ [Dir, P, nm, C, _] and (nm != "..") (5)

[Dir, D1, ".", _, D2] ⇒ D1==D2 (6)

[Dir, _, "..", _, P] ⇒ is_ancestor(ROOT, P) (7)

Finally, Invariant 7 checks that a directory update does not cause cycles. Invariants 4 and 5 do not prohibit cycles. For example, suppose that the file system allows the command "mv /a /a/b" to complete successfully. This update would be allowed by the Invariants 4 and 5, but it would create a disconnected cycle consisting of a and b. Invariant 7 checks for cycles when a directory's parent entry (the ".." entry) is updated. It ensures that the chain of parent directories eventually reaches the root directory, or a cycle is detected. The is_ancestor() primitive operates on the Recon metadata caches described in Section 4.

3.4 Btrfs Inode and Directory Entries

Metadata structures in btrfs are indexed by a 17-byte key consisting of the tuple (objectID, type, offset). ObjectID is roughly analogous to an inode number in ext3. The type field determines the type of the structure, and the meaning of "offset" depends on the type. Each key is unique within a btrfs tree, so the unique (type, id) pair for our change records consists of (type, (tree id, objectid, offset)).

A btrfs consistency property is that the inode associated with a directory item (that is, a btrfs directory entry) has a directory mode (row F in Table 3). An invariant derived from this property is that when we add a new directory item, there must exist an appropriate inode item after transaction commit. We can represent this as:

[DIR_ITEM, (T, I, _), _, ϕ, _] ⇒
  exists(T, I, INODE_ITEM, 0) and
  ISDIR(get_item(T, I, INODE_ITEM, 0).mode)

The left hand side matches a directory item within snapshot tree T and objectid I that is being newly created. This invariant asserts that 1) there is a matching inode item, and 2) its mode is of directory type. The exists() primitive returns true if the given item can be found in tree T, and the get_item primitive obtains the contents of the item, allowing us to check the mode. These primitives operate on the Recon metadata caches.

4 Implementation

We use the Linux device mapper framework to interpose on all file system I/O requests at the block layer, as shown in Figure 1. On a metadata block read, recon_read caches the block in the Recon read cache. This cache allows accessing the disk or the pre-update file-system metadata state efficiently. Its contents are trusted because its blocks have been verified previously. On a metadata block write, recon_write caches the updated block in the Recon write cache. The write cache may contain corrupt data and thus any code accessing this cache must perform careful validation. Both caches also store block-specific information such as the block-type map. Similar to a file system buffer cache, neither Recon cache persists across reboots.

4.1 Commit Process

At commit, our framework requires that 1) all transaction blocks must have been recorded using recon_write, and 2) recon_commit is called before the commit block reaches the disk. We can record blocks and detect commit either from the transaction subsystem (transaction-layer commit) or at the block layer (block-layer commit). With transaction-layer commit, the file system's transaction commit code is modified to invoke recon_write on the updated metadata blocks, and invoke recon_commit before writing the commit block. This method is simpler to implement, but it makes us dependent on the transaction layer code, such as JBD in ext3. In particular, it does not allow us to verify the ext3 checkpointing process.

With block-layer commit, recon_write could be invoked on all block writes. The challenge is to separate metadata blocks from data blocks because we do not want to cache every data block. However, we can only identify newly allocated metadata blocks at commit, making them hard to distinguish from data on each write. Fortunately, for ext3, metadata blocks are written to the journal, and thus we can ignore blocks that are not journaled. This approach requires interpreting journal writes at the block layer, which also helps detect commit. While this implementation is more complicated, it removes any dependency on the journaling code. For btrfs, metadata writes can be easily distinguished because they are directed to designated regions on disk called btrfs chunks. Btrfs commits occur when the superblock is written, which is easy to detect because the superblock is in a known location.

We have implemented both transaction-layer and block-layer commit, but currently we have only evaluated the transaction-layer commit implementation.

4.2 Cache Pinning and Eviction

We control the amount of memory used by the Recon caches with a simple LRU mechanism for replacing blocks from the read cache when it grows beyond a user-configurable limit. All read cache blocks are pinned during recon_commit processing to simplify implementation. We expect that recon_commit will run quickly because the blocks needed for commit processing have likely been read by the file system recently and so they will not need to be read from disk to populate the read cache. We pin the Recon write cache for the duration of the transaction because we will need these blocks for checking invariants. This approach is similar to the ext3 file system pinning its journal blocks for performance. However, we could unpin a block once it reaches disk, e.g., the journal in ext3.

After commit, the contents of the write cache are merged into the read cache, thus updating Recon's view

FS Recon API	Invoked on	
references	Read	provides type and id information for data structures in referenced blocks
process_write	Commit	provides type and id information for newly allocated metadata blocks
process_txn	Commit	generates change records
txn_check	Commit	checks invariants using change records and metadata read/write caches

Table 4: File-system specific Recon API

of file-system state, and the write cache is cleared. At this point, we can unpin the read cache because all the blocks in the cache are on disk (e.g., either in the journal or the checkpointed location in ext3). However, our transaction-layer commit implementation for ext3 does not track the location of blocks in the journal. To avoid evicting a block that may be in the journal, we keep a list of most recently updated blocks in the read cache. This list contains as many blocks as it takes to fill the journal and we pin these blocks. Once a block is evicted from this list, it must have been checkpointed, or else it would have been overwritten in the journal, and so we can unpin it.

4.3 File-System Specific Processing

Recon invokes file-system specific API functions for metadata interpretation and invariant checking, as shown in Table 4. The *references* function is invoked by recon_read to parse a metadata block and create block-type mappings for pointed-to blocks. This function is also used to distinguish between data and metadata on the read path.

The rest of the functions in Table 4 are invoked by recon_commit. The *process_write* function is similar to the references function but invoked on all the blocks in the write cache (i.e., each updated or newly allocated metadata block). This function must validate the updated blocks by checking that any pointers, strings and size fields within the block have reasonable values so that further processing is not compromised. Recon ignores unknown blocks and only processes updated blocks whose types are known. As unknown blocks become known, they are added to the queue of blocks being processed. At the end of write processing, if any unknown blocks remain, Recon signals a reachability invariant violation, as discussed in Section 2.3.

Once the block and data types within blocks are known, the *process_txn* function compares updated data structures with their previous versions to derive a set of change records. The previous version of a data structure is uniquely determined by the (type, id) pair of the change record. In ext3, the type is determined by block type and the id is typically an inode number or a block number. In btrfs, the type and id are determined by the tree and the key, as discussed in Section 3.4.

While the process of comparing data structures is clearly file-system specific, we found two common cases. When data structures have fixed size, such as inodes in ext3 and most items in btrfs, we use a simple byte-level diff that is driven by tables that describe the layout of the data structures. These tables are generated from the data structures using C macros. When data structures themselves contain sets of smaller items, such as directory entries in ext3, or extent items in btrfs, we use a set-intersection method to derive three sets consisting of new items, deleted items and modified items. Change records can be generated from these sets, using the identity of the containing item (e.g., directory inode) and some key as field name (such as the "name" for directory entries).

The *txn_check* function implements invariant checking as described in Section 3 with examples.

4.4 Handling Invariant Violation

The final problem for an online consistency checker like Recon is dealing with invariant violations. It is important to ensure that recovery from a violation is correct and so the safest strategy is to disable all further modifications to the file system to avoid corruption. The file system can then be unmounted and restarted manually or transparently to applications [26]. In this case, the file system is not corrupt but may have lost some data. If the ability to create a snapshot (e.g., a btrfs snapshot) is available, then a snapshot could be created immediately, the problem reported, and then we could continue running the file system. It is important to isolate the snapshot from the buggy file system, e.g., by directing all further writes to a separate partition. In this case, data is preserved but the file system may be corrupt. Finally, it may be possible to repair file system data structures dynamically [8].

5 Evaluation

In this section, we evaluate our Recon implementation for ext3 in terms of its 1) complexity, 2) ability to detect metadata corruption at runtime, and 3) its performance impact. Currently, we are finishing our btrfs implementation, and we plan to evaluate it in the near future.

5.1 Completeness and Complexity

We have implemented all of the checks performed by the e2fsck file system checker, as encapsulated by the SQCK rules, for the mandatory file system features. Overall, we need only 31 invariants (vs 101 SQCK rules) because some properties are easier to verify at runtime. For example, a large number of fields in the superblock and block group descriptors are protected with the simple invariant that they should not be changed by a running file system.

We also avoid explicit range check invariants in several cases because they are naturally embedded in other invariants that must check for setting or clearing of bits in bitmaps. There are a small number of properties on optional features that we do not check, such as OS-specific fields in inodes and the extended attributes ACLs.

Our entire system consists of 3.8k lines of C code (kLOC), as measured by the cloc [7] tool. Of these, 1.5 kLOC are in the generic framework which can be reused across file systems, 1.5 kLOC are for interpreting the ext3 metadata, and only 0.8 kLOC are involved in checking the invariants. Our dependence on the journal checkpointing code adds another 311 lines. The code required to do the checking is simpler than the file system code for several reasons. First, within the thread checking a transaction, we do not need to worry about concurrency, as the buffers we are examining are under the control of the journal. In contrast, the file system needs to be servicing multiple client threads. Second, the implementation of each invariant check is independent of the other checks because each rule uses its own data structures to keep track of properties that must be verified. Finally, the implementation of each rule is usually quite simple, requiring several lines of C to accumulate the necessary data and a few more (often just a single boolean expression) to verify.

5.2 Ability to Detect Corruption

Evaluating resiliency against metadata corruption is tricky. To best represent real-world corruption scenarios, we would either inject subtle bugs in the file-system or reproduce known bugs. However, subtle bugs (i.e., bugs not easily found in a heavily-used file system) are hard to design or reproduce. Reproducing known bugs is difficult as they often depend on specific kernel versions, combinations of loadable modules, concurrency levels, or workloads. Instead, we settled for deliberately injecting corruption of bytes within metadata blocks. This mimics the corruption that could result from several types of bugs (e.g., setting values in arbitrary fields incorrectly) both within the file system or in the overall kernel. We injected both type-specific corruption, where we target specific metadata block types and fields, and fully random corruption where we corrupt a sequence of 1 to 8 bytes within some number of blocks in a transaction.

Setup We compare Recon against e2fsck by corrupting metadata just before it is committed to the journal. We begin each corruption experiment by creating and populating a fresh file system, to ensure that there are no errors initially. Next, we start a process that creates a background of I/O operations (specifically we run a kernel compile and clean, repeatedly). The corruptor then sleeps for 20-90 seconds, wakes up, and performs the requested corruption (type-specific or random). We record

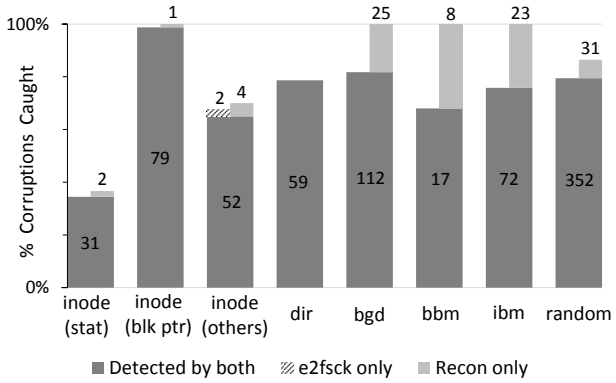

Figure 3: Comparison of corruption detection accuracy

the corruption performed and whether or not Recon detected it. Next, we allow the transaction to commit, and then immediately prevent any future writes. This step ensures that the corruption is limited to the bytes that we selected, rather than the result of the file system acting further on corrupt data. Next, we unmount the file system, run e2fsck on it, and record whether it found and repaired any errors. Finally, we run e2fsck a second time to see if the file system is clean after the repairs, and then reboot the system for the next experiment. For these experiments, we use a 4 GB file mounted as a loop device for our file system. This simplified the restoration of the file system following each corruption experiment.

Our corruption framework can only corrupt blocks that the file system is already modifying in some transaction. In particular, we never corrupt the superblock since the running file system never includes writes to it. We do not consider this to be a serious limitation to our test results since nearly all superblock corruptions would be trivially detected by Recon. Specifically, Recon protects most fields in the superblock with the invariant that they should not be modified at all, which is very easy to check.

Results Figure 3 summarizes the results of our corruption experiments. We show a wide bar and two stacked bars for each type of metadata corruption and random corruption. The wide bar shows the percent of corruptions (Y axis) that were caught by both e2fsck and Recon. The stacked bars show the percent of corruptions that were detected by only one checker. Numbers in the bars show the absolute number of corruptions detected.

For inodes, we present 3 sets of bars, representing different types of inode fields. The first group includes fields that are reported by "stat", the second group consists of all the block pointer fields, and the third group consists of everything else. Our coverage is nearly identical to e2fsck in all cases. Many of the inode stat fields are unrelated to file system consistency (e.g. the timestamps and userids) and are permitted to change arbitrarily, making it hard to detect corruption with either checker. However, both check-

ers are effective at catching corruption of block pointers. Recon achieves 100% in this case because it checks all inodes in a block being written to disk while e2fsck ignores unused inodes. Although file system consistency is not affected by changes to unused inodes, it is still useful to detect this corruption because it indicates a bug in the system. For the final set of inode fields, e2fsck detects an invalid flag setting that Recon does not check in two runs, while Recon catches corruption of some unused inode flags and a corruption of the dir_acl field that appears valid when checked by e2fsck after the fact in four runs.

For directory entries (dir), both checkers detect the same corruptions, with neither checker detecting corruption of the name field. For the other metadata types, Recon is more effective than e2fsck at detecting corruption, largely because it is able to take other runtime behavior into account. For example, Recon achieves 100% detection for block group descriptor (bgd) corruption because most of these fields should not be changed by a running file system. Once corruption has reached the disk however, it is not always possible to distinguish the correct values from corrupted, but still valid, values. Similarly, Recon detects 100% of the block and inode bitmap (bbm and ibm, respectively) corruptions while e2fsck has a lower detection rate because it does not check unused parts of metadata blocks. For example, e2fsck does not check bits in the inode bitmap for non-existent inodes, or bits in the block bitmap for uninitialized block group descriptor table blocks. Recon's higher coverage on specific metadata fields leads to higher coverage for fully random corruption as well. We expect that adding the final set of ext3 invariants for OS-specific inode fields and extended attributes will help us detect all ext3 structural consistency violations. However, neither checker can achieve 100% accuracy because some of the corruptions hit fields unrelated to structural consistency.

After e2fsck performs repair, it still detects errors in 28 out of 731 cases (3.8%), when it is run a second time on the "repaired" file system. Two of these failures occurred after a single byte was corrupted in a single metadata block. In our experiments, we unmount the file system and check it with e2fsck immediately after the corrupted transaction is committed to the journal. In reality, it is likely that the file system would continue operation with bad data for some time, making the chances of successful repair even lower. In these cases, Recon's ability to prevent corruption from reaching the on-disk metadata is particularly valuable.

5.3 Performance

Setup All performance tests were done on a 1 TB ext3-formatted file system on a machine with 2GB total RAM and dual 3 GHz Xeon CPUs. We used the Linux port of FileBench (version 1.4.8.fsl.0.8) with the application emulation workload personalities[4]. We included the Networkfs personality, which supports a more sophisticated file system model, with a custom profile configured to match the metadata characteristics from a recent study of Windows desktops[17]. For Fileserver, we reduced the default file size to 32k to increase the metadata to data ratio in the file system. In all other cases, we used default parameter settings. Table 5 summarizes the basic characteristics of our benchmarks.[5] The metadata load varies widely across the benchmarks, spanning the range of Recon cache sizes, causing misses in the cache. In particular, the Fileserver benchmark uses over 25k directories. The metadata consumed by directory entry blocks alone is greater than 100MB. The inodes for the directories and files would consume approximately 70MB if they were stored compactly, but ext3 distributes allocation across different block groups, so unused inodes add to the metadata overhead. While the Networkfs benchmark involves more file data, the total number of files is lower because of the larger file size distribution.

Personality	Settings	Data Size
Webserver	nfiles=250k	3.9 GB
Webproxy	nfiles=500k	7.8 GB
Varmail	nfiles=250k	3.9 GB
Fileserver	nfiles=500k, filesize=32k	15.6 GB
MS-Networkfs	based on [17]	19.9 GB

Table 5: Benchmark Characteristics

The benchmarks are run for one hour for all workloads to ensure that we capture steady-state behavior with Recon. We report the performance of Recon compared to native ext3 for both the initial benchmark setup, which involves heavy metadata writes (Table 6), and the actual workload execution (Figure 4).

Our current transaction-layer commit implementation (described in Section 4) cannot evict blocks from our metadata cache that have not yet been checkpointed to the file system. Thus, the metadata cache size must be larger than the journal size. However, any memory consumed by Recon's metadata cache reduces the memory available for the file system cache by the same amount because Linux implements a shared page cache. We present results for three different cache/journal sizes, for both native and Recon performance. FileBench emulates workloads using a variety of random variables for file and operation selection. Thus, there is natural performance variation across runs. Since this is representative of behavior "in the wild", we report the average of 5 runs with error bars. All tests are done with cold caches on a freshly booted system.

[4]The OLTP personality did not work in the version we obtained.

[5]The full profile used in the experiments is available at http://csng.cs.toronto.edu/publications/260/get?file=/publication_files/210/recon-fast2012-workloads.tgz

	Cache=64MB, Journal=32MB			Cache=128MB, Journal=64MB			Cache=256MB, Journal=128MB		
Setup (seconds)	**Ext3**	**Recon**	**Ratio**	**Ext3**	**Recon**	**Ratio**	**Ext3**	**Recon**	**Ratio**
Webserver	2171.0±42.8	2903.2±45.7	133.7	1722.0±77.4	1668.6±36.7	96.9	1405.6±24.4	1340.2±29.6	95.3
Webproxy	229.4±26.0	323.0±24.3	140.8	212.8±13.5	243.4±23.5	114.4	227.2±19.5	224.4±24.0	98.8
Varmail	110.2±11.4	110.8±4.4	100.5	118.6±12.3	113.8±16.2	96.0	109.4±9.5	123.0±5.0	112.4
Fileserver	13728.5±694.2	17705.8±413.5	129.0	11487.2±849.8	12906.8±1316.8	112.4	9785.6±491.6	10374.8±928.8	106.0
Networkfs	2096.8±140.4	2113.8±119.2	100.8	1757.4±70.2	1893.0±73.0	107.7	1651.8±113.8	1719.4±31.5	104.1

Table 6: Setup time for benchmarks (lower is better)

Results During the benchmark setup, when many files are being created, there is a significant cost to Recon, particularly for small cache sizes. The dominating factor is I/O time for metadata cache misses because file creation quickly and repeatedly touches the entire working set of metadata. However, as the cache size increases, the impact is rapidly reduced. With a 128MB metadata cache, the added overhead of Recon is within the experimental error of ext3's native performance. The impact of Recon is less noticeable during normal benchmark operations. With our smallest metadata cache size (64MB), there is a worst case overhead of only 15% for Fileserver, which is generally reduced as the cache size increases. The one exception to this trend is the Networkfs personality (ms_nfs in Figure 4), where performance degrades with an increasing Recon cache size. We believe this is the result of memory pressure, as our increased metadata cache size decreases the amount of memory available to the file system buffer cache. Overall, a 128MB metadata cache with a 64MB journal gives the best results for all workloads, with only 8% degradation on average. In most cases, file system throughput with Recon is within the margin of error of ext3 performance. Given the growth in main memory sizes, these are quite modest memory requirements for the reliability benefits that Recon can deliver.

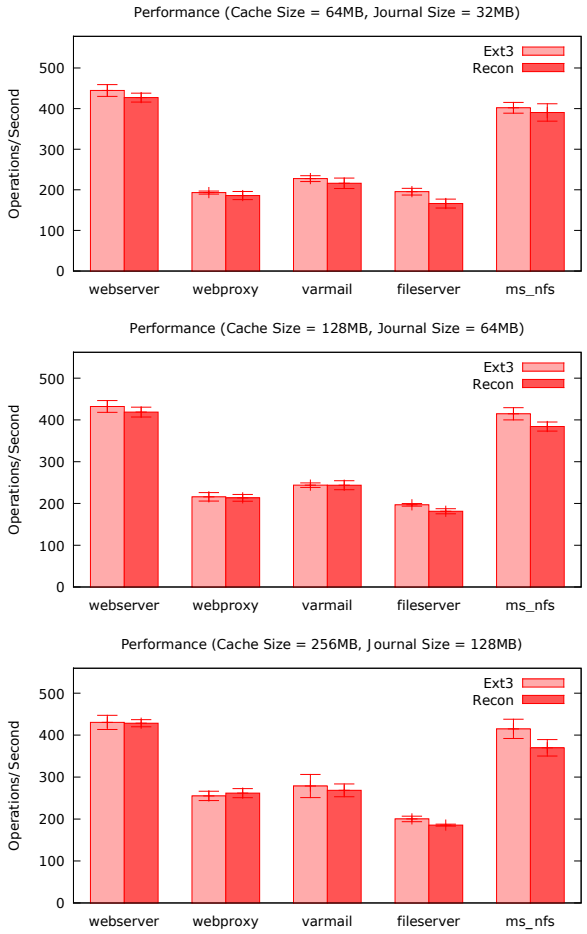

Figure 4: Performance on FileBench workloads for varying metadata cache sizes

6 Related Work

We discuss several areas of research that are closely related to this work, including methods for 1) handling file system bugs, 2) checking file system consistency, 3) interpreting file system semantics and verification.

6.1 Handling File System Bugs

File system bugs can be detected statically or at runtime. Bug finding tools, based on model checking [29, 31] and static analysis [21], have revealed scores of bugs in a variety of file systems. However, these tools cannot be relied upon to identify all bugs because they need to perform exhaustive evaluation. Furthermore, even when a bug is known, a bug fix may not be easily available, or easy to deploy in live systems [1]. These limitations can be addressed by tolerating bugs at runtime.

EnvyFS [3] applies N-version programming for detecting file system bugs. It uses the common VFS interface to pass each file system request received by the VFS layer to three child file systems. The results are then compared and the majority result is returned. EnvyFS avoids storing 3 data copies by using a customized single-instance store. Although EnvyFS is able to detect and in some cases repair errors introduced in child file systems, the run time overheads are significant because the operations must be issued to at least two file systems and the results compared before an answer is returned. Also, subtle differences in file system semantics can make it hard to compare results.

Membrane [26] proposes tolerating bugs by transparently restarting a failed file system. It assumes that file system bugs will lead to detectable, fail-stop crash failures. However, inconsistencies may have propagated to the on-disk metadata by the time the crash occurs. Our approach is complementary to Membrane, rather than wait-

ing for the file system to crash, a restart could be initiated when Recon detects an inconsistent transaction.

6.2 Checking File System Consistency

SQCK [11] expresses the many complex checks performed by e2fsck as a set of compact SQL queries. It improves upon the repairs done by e2fsck by correcting the order in which repairs were performed and by using redundant file-system metadata ignored by e2fsck.

Chunkfs proposes reducing the consistency check time by breaking the file system into chunks that can be checked independently [13]. While this idea is appealing, unfortunately the chunks are not independent and thus cannot be checked truly independently. Specifically, pathnames can span chunks, and Chunkfs uses cross-chunk references to handle hard links and files that are larger than chunks or need allocation across chunks.

ZFS provides the ability to scrub disks and repair corrupt blocks that have redundant copies [4]. Scrubbing can detect latent hardware errors but does not necessarily detect software bugs, e.g., if the block has a consistency error but passes the checksum. NetApp filers can run some phases of the wafliron check program on an online system, but this process is resource intensive and time-taking.

6.3 File System Semantics and Verification

Semantically-smart disks use probing to gather detailed knowledge of file system behavior [24]. This knowledge is used at the block interface to transparently improve performance or enhance functionality, such as by implementing track-aligned extents and secure delete. This work builds on several ideas from semantically-smart disks.

The XN storage system of the Xok exokernel is designed to protect library file systems that manage their own disk blocks [15]. XN uses a file-system specific function called own(), similar to the Recon references() function, that returns the blocks controlled by a meta-data block. This function allows XN to verify that a file system can only access blocks that are allocated to it. XN can also use a file-system specific function called reboot() that traverses the entire file-system tree and detects whether the file system is crash consistent. This work shows that file-system consistency can be verified at runtime efficiently. File systems must use an extended block interface (e.g., allocate, read, write, deallocate) and provide block type information to XN and which allows easier verification, while Recon only requires the basic block interface (e.g., read, write) and infers file system information. Also, XN protects file systems from each other and may allow a file system to corrupt itself, while our focus is on protecting the file system from itself. Similar to XN, a type-safe disk extends the disk interface by exposing primitives for block allocation [23], which helps enforce invariants such as preventing accesses to unallocated blocks.

There has been significant work on discovering program invariants by capturing variable values at key points in a program to repair data structures [8] and to patch buggy deployed software [18]. We plan to apply these methods to learn file-system invariants and repair updates that cause invariant violations. Our work is influenced by runtime verification, a technique that applies formal analysis to the running system rather than its model [25, 5].

Our system can be viewed as a firewall with a set of rules that help protect disks from accesses that could compromise file-system integrity. Defining and implementing these rules in a high-level language, such as the Linux iptables rules [22], is an avenue for future work.

7 Conclusions and Future Work

The Recon system protects file system metadata from buggy file system operations. It uses two key ideas, using *commit points* to verify *consistency invariants*. Modern file systems aim to ensure file system consistency at commit points. Consistency invariants are declarative statements that must be satisfied at these points before data is committed or else the file system may get corrupted. We reuse the consistency rules used by a file system checker to derive the invariants. As a result, Recon detects random corruption at runtime as effectively as the file system checker. It has low overhead because the data it interprets has likely been recently accessed by the file system.

A system that checks the file system is easier to implement correctly than the file system itself. When checking a transaction, we do not need to worry about concurrency because the buffers we are examining are under our control. In contrast, the file system needs to be servicing multiple client threads. Also, each invariant is independent because it uses its own data structures to keep track of the properties that must be checked, and we find that the implementation of each rule usually quite simple. The bulk of the complexity lies in interpreting metadata structures. We plan to develop a systematic way to describe and interpret these structures.

While an offline checker can only make decisions based on the current file system state, Recon can also observe file operations in progress. We plan to investigate whether this allows detecting certain operational bugs unrelated to file system consistency, e.g., updates to userid fields.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Junfeng Yang, for many insightful comments. We also thank Vivek Lakshmanan, who provided several insights that helped start this project, and several members of the SSRG group at University of Toronto for their feedback on initial versions of the paper. This research was supported by NSERC through the Discovery Grants program.

References

[1] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: automatic rebootless kernel updates. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)* (2009), pp. 187–198.

[2] BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., GOODSON, G. R., AND SCHROEDER, B. An analysis of data corruption in the storage stack. *Transactions of Storage 4*, 3 (2008), 1–28.

[3] BAIRAVASUNDARAM, L. N., SUNDARARAMAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Tolerating file-system mistakes with envyfs. In *Proceedings of the USENIX Technical Conference* (June 2009).

[4] BONWICK, J., AND MOORE, B. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[5] CHEN, F., AND ROŞU, G. Mop: an efficient and generic runtime verification framework. In *Proceedings of the ACM OOPSLA* (2007), pp. 569–588.

[6] CUSTER, H. *Inside the Windows NT File System*. Microsoft Press, 1994.

[7] DANIAL, A. CLOC – Count Lines of Code. http://cloc.sourceforge.net/.

[8] DEMSKY, B., AND RINARD, M. C. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering 32*, 12 (2006), 931–951.

[9] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems 18*, 2 (2000), 127–153.

[10] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2007), pp. 293–306.

[11] GUNAWI, H. S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[12] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (Nov. 1987).

[13] HENSON, V., VAN DE VEN, A., GUD, A., AND BROWN, Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)* (2006).

[14] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference* (1994).

[15] KAASHOEK, F. M., ENGLER, D. R., GANGER, G. R., BRICENO, H. M., HUNT, R., MAZIKRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., , AND MACKENZIE, K. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (1997), pp. 52–65.

[16] MASON, C., AND ET AL. Btrfs. http://btrfs.wiki.kernel.org.

[17] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2010).

[18] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S. P., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. C. Automatically patching errors in deployed software. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2009), pp. 87–102.

[19] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Model-based failure analysis of journaling file systems. In *Proceedings of the IEEE Dependable Systems and Networks (DSN)* (2005), pp. 802–811.

[20] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Iron file systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2005), pp. 206–220.

[21] RUBIO-GONZÁLEZ, CINDY, GUNAWI, S., H., LIBLIT, B., ARPACI-DUSSEAU, H., R., ARPACI-DUSSEAU, AND C., A. Error propagation analysis for file systems. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI)* (2009), pp. 270–280.

[22] RUSSELL, R. Iptables. http://en.wikipedia.org/wiki/Iptables.

[23] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)* (2006), pp. 15–28.

[24] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2003), pp. 73–88.

[25] SOKOLSKY, O., SAMMAPUN, U., LEE, I., AND KIM, J. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science 144* (May 2006), 91–108.

[26] SUNDARARAMAN, S., SUBRAMANIAN, S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. Membrane: Operating system support for restartable file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2010).

[27] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference* (1996), pp. 1–14.

[28] TWEEDIE, S. C. Journalling the ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo* (May 1998).

[29] YANG, J., SAR, C., AND ENGLER, D. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Operating Systems Design and Implementation (OSDI)* (2006).

[30] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006), pp. 243–257.

[31] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (2006), 393–423.

[32] ZHANG, Y., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. End-to-end data integrity for file systems: a ZFS case study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2010).