

Verifying File System Consistency at Runtime

Abstract

Existing file-system reliability methods, such as checksums, redundancy, or transactional updates, provide limited defenses against file-system bugs that cause disk corruption. The existing workarounds, based on using backups or repairing the file system, are painfully slow. Worse, the recovery is performed much after the error occurred, and thus may result in further corruption and data loss.

We present a system 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. By checking each transaction before it commits, we can minimize the damage caused by buggy file systems.

The major challenges to this approach are specifying invariants, and correctly interpreting file system behaviour without relying on the file system code. Our prototype system, called 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 and btrfs file systems in this framework. For ext3, Recon can detect random as well as targeted file-system corruption at runtime as effectively as the offline e2fsck file-system checker, with low performance overhead.

1 Introduction

It is no surprise that file systems have bugs, as shown by several studies [22, 29, 31]. Modern file systems are designed to support different environments ranging from smart phones to high-end servers. They must support concurrent operation and deliver high performance. Further, they must handle a large number of failure conditions, while preserving consistency and data integrity. Ironically, the resulting complexity leads to bugs that can be hard to detect, even with heavy testing. File system bugs can lead to silent corruption of disk data [22, 21]. Such corruption is hard to detect and may cause data loss, the return of corrupt data to the user, persistent application crashes, or even worse, security exploits [30].

File-system corruption can be caused by crash failures, storage hardware errors, file-system bugs and memory corruption. Storage systems have primarily focused on the first two problems. Crash failure recovery can be per-

formed using transaction update methods, such as journaling [15] and copy-on-write [17], and soft updates [12]. Hardware errors can be handled by using checksums and redundancy for error detection and recovery [8, 13].

As these sources of error are diminished, corruption and data loss induced by file system bugs and memory corruption will become more prominent. Unfortunately, handling these problems is much more challenging. For example, modern file systems such as Sun's ZFS have been carefully designed to handle a wide range of disk faults. However, the machinery used for protecting against disk corruption (e.g., checksums, replication and transactional updates) does not help with memory corruption, which can still cause the return of corrupt data to applications and system crashes [32]. Today, these risks hinder both file system development and the adoption of new and improved file systems.

When a file system bug corrupts data, 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 backups and snapshots, but these can result in significant downtime and loss of recent data.

Another option is to use an offline consistency check mechanism, provided by most file systems, for restoring file system consistency. While a consistency check can detect most failures, it raises two problems. First, the entire disk needs 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 disk seek time [16]. Second, the consistency check is run after the fact, often after a sudden power loss or system crash or even less frequently with journaling file systems. Thus an error may propagate and cause significant damage to the on-disk image between the time it is triggered and the time the check is performed. Since a consistency checker lacks past history of file operations, correct repair may simply be impossible.

Our aim is to prevent data corruption in the face of *arbitrary* file system bugs and memory corruption. This paper presents a system called Recon that observes file system behavior and guards against writes that manifest symptoms of a buggy file system so that corruption does not propagate to the storage media. Specifically, Recon verifies that the file system always preserves metadata consistency. Metadata is more vulnerable to corruption by file system bugs since the file system does not usually manipulate the contents of user data blocks. A small corruption

in metadata can also result in significant loss of user data, since a file system operating on incorrect metadata may overwrite existing data or render it inaccessible.

Our approach addresses three questions: 1) *when* should the consistency properties be checked, 2) *what* properties should be checked, and 3) *how* should they be checked.

The in-memory copies of metadata may be temporarily inconsistent during file system operation, and so it is non-trivial 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 copy-on-write file systems are already designed to handle crash failures using transactional update 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 itself to be consistent, and hence transaction boundaries serve as convenient vantage points for verifying consistency properties. A violation of these properties at these points indicates a file system bug or a memory corruption.

Second, identifying the correct consistency properties is challenging because the behavior of the file system is not formally specified. Fortunately, the source code of the file system consistency checker provides a comprehensive set of consistency properties. For example, Gunawi et al. found that the Linux `e2fsck` program checks 121 properties that are common to both ext2 and ext3 file systems, as well as some additional properties related to the ext3 journal and optional features [14].

Unfortunately, consistency properties are *global* statements about the file system. For example, a simple check implemented by `e2fsck` is that the deletion time of *all* used inodes is zero. Determining the in-use status of each inode, and checking the deletion time of each used inode would be infeasible at every transaction commit point. 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.

The novelty of our solution lies in transforming each consistency property to a *consistency invariant*, which is a local assertion that must hold for a transaction to preserve the file system’s consistency properties. 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. This invariant can be checked by examining only the updated blocks, i.e., the updated pointer block, the allocated block and the updated block bitmap must all be part of the same transaction.

Our approach relies on an inductive argument. It as-

sumes 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 provides more details about our assumptions.

We also rely on the ability to convert consistency properties requiring global information into invariants that can be checked using only limited, “local” information, as described in the previous example. The insight is that such a transformation should be possible because a working file system must also try to ensure that each of its operations maintains the consistency properties. As a result, if a consistency invariant requires a global scan, the file system itself would need to perform a similar scan to access the same data to ensure consistency. In other words, our invariant checking should not require much more data than the file system itself needs for its operations. As described later, our experience with the Linux ext3 file system shows that a limited subset of the metadata is needed to check consistency invariants at each transaction.

Lastly, checking invariants is challenging because it requires understanding the file-system data format and the semantics of transactions. However, transactions need to be interpreted outside the file system since the entire file system is suspect. We resolve this conflict by providing a framework for file-system specific metadata interpretation and invariant checking. The framework operates at the block layer and allows interpreting file-system specific information using a separate metadata cache, similar to the design of the storage system in the Xok exokernel [18]. A separate cache is maintained because the file system cache is untrusted and because it allows checking the invariants efficiently.

Our current implementation of Recon shows the feasibility of interpreting metadata and writing consistency invariants for two radically different file system designs, the widely used Linux ext3 and the recently deployed Linux btrfs file systems. Recon supports metadata interpretation for btrfs, but currently we have only implemented a small number of btrfs invariants, and so our evaluation focuses on the ext3 implementation. Recon checks ext3 invariants corresponding to most of the consistency properties checked by the `e2fsck` program. Thus, it can detect type-specific and random file-system corruption as effectively as the offline checker, with low memory and performance overhead. At the same time, our approach does not suffer from the serious limitations of offline checking described earlier because corruption is detected immediately. Furthermore, Recon can detect certain types of data corruption that do not affect file system consistency but are nevertheless indicative of buggy file system or kernel code.

The rest of the paper describes our approach in detail

and presents the results of our initial evaluation.

2 Fault Model

Our goal is to preserve file-system metadata consistency in the presence of arbitrary file-system bugs and memory corruption. We make 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 transaction inconsistency and the 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 system.

Second, the Recon code is implemented at the block device layer in the Linux kernel and invoked by the transaction commit code for simplicity. For example, the ext3-specific code of Recon is invoked by JBD, the journaling block device layer used by ext3, when JBD commits transactions to the journal device. We depend on the correctness of commit code as well as the journal checkpointing code. It may appear that consistency could be checked at journal checkpointing rather than commit, reducing our dependence on the checkpointing code. However, this is not feasible because if a crash occurs after a commit, the journal or disk contents could be corrupt, and we would have lost our metadata cache and thus our ability to check consistency during recovery. JBD has much more functionality, such as buffer management, and we do not depend on its correctness. For example, if JBD corrupts or loses some buffers, Recon would detect a consistency violation. A hypervisor implementation would provide stronger guarantees against arbitrary kernel memory corruption. It would also avoid dependence on all JBD code except the checkpointing code (it would avoid any dependence on btrfs code, because btrfs does not perform checkpointing).

Finally, our inductive assumption about metadata consistency before each transaction (discussed in Section 1) requires correct functioning of 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. The btrfs code provides such functionality [10]. For ext3, this functionality could be implemented at the block layer [13]. If these assumptions are not met, offline checking and repair could be used as a last resort.

3 Consistency Invariants

We first discuss when file-system consistency properties should be checked. Then, we briefly discuss the properties

that are checked by the ext3 and the btrfs file-system consistency checker. Finally, we describe how consistency invariants can be derived from consistency properties.

3.1 Commit Points

We check file-system consistency properties at commit points at which the file system itself aims to ensure metadata consistency. These commit points are available in file systems that provide crash consistency using a transactional update model. Examples include journaling file systems that use a write-ahead log [1, 3, 4], and copy-on-write file systems that never overwrite valid data [17, 8, 10]. Our approach applies to both logical journaling (log records describe the logical operation being performed) and physical journaling (log records contain copies of modified blocks). In the latter case, block differencing allows deriving the logical updates in the transaction.¹

3.2 Consistency Properties

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 (not repairing) inconsistencies. Since we have applied our approach to the ext3 and the btrfs file systems, we provide an overview of the consistency rules for these file systems below.

The SQCK system [14] encapsulates the 121 checks of the ext3 fsck program in a set of declarative (SQL) queries. Although there is a close correspondence between SQCK queries and e2fsck checks, some SQCK queries combine multiple checks. Table 1 provides a breakdown of the number of rules checked by SQCK for different file-system data structures. We show 101 rules, 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). Certain 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 in the file system form a connected tree.

We have done a similar classification of the rules checked by the btrfs checker, as shown in Table 2. Btrfs is an extent-based, btree file system [10]. It uses btree node extents to store pointers to child nodes or leaves, and btree leaf extents to store items that contain the file system metadata structures (e.g., inode item, directory item, etc.). Btrfs uses a copy-on-write transaction model for

¹Implementing consistency invariants for soft update file systems [12] that provide consistency after each write but no commit points should be possible but will likely be more complicated.

Datatype	#rules
Within superblock	23
Within block group descriptors (BGD)	5
Within a single inode	28
Within a single directory	14
Between inode and directory entries	5
Between inode and its block pointers	2
Between inode, inode bitmap, orphan list	3
Between block bitmap and block pointers	5
Between block, inode bitmaps, BGD table	3
Between directories	4
Bad blocks inode	7
Extended attributes ACL	2

Table 1: Number of Ext3/SQCK rules by datatype

updates and for supporting file-system snapshots. Each snapshot is a separate, copy-on-write file system, called a file-system tree. Extent allocation information is maintained in an extent btree, which serves the same purpose as ext3 block bitmaps. Checksum information is maintained separately in an extent-level checksum btree. The roots for all the btrees are maintained in another btree called a root tree.

Btrfs has three features that make it easier to check file system consistency. First, the use of a single btree data structure for all file system data reduces the number of checks that have to be implemented. Although the btrfs checker is still a work in progress (e.g., it performs no repair), currently it uses only 30 rules for detecting inconsistencies, as shown in Table 2. The next section discusses some of these rules. Second, extents are self identifying. Each extent has a header that stores the type (node or leaf) and the physical location of the block, thus simplifying the effort needed to interpret transactions. Third, btrfs uses back references extensively, e.g., a child extent has a reference to its parent, and inodes have references to each of their directory entries (i.e., hard links). Back references provide redundancy, enabling additional checking.

However, btrfs snapshots complicate consistency checking. Each snapshot file-system tree is self consistent, and so a naive consistency check must traverse each file-system tree separately. If many unmodified, copy-on-write snapshots are created for the default file-system tree, the consistency check time would be proportional to the number of snapshots rather than the total amount of file system metadata. The btrfs checker uses an elegant, although complicated, method to verify the consistency of all file-system trees while traversing each extent once. The basic idea is to traverse a shared node and its sub-tree once and aggregate any sub-tree state that is needed for checking rules. When this shared node is reached from another file-system tree, its state is reused and merged with the state of the second file-system tree. The merg-

Datatype	#rules
Within metadata extent (block)	2
Between parent and child extents	3
Between extent tree and extents	3
Within an extent item in extent tree	2
Between inodes and file system trees	2
Between inode and directory entries	4
Between inodes, inode refs and dir. entries	2
Within directory entries	1
Between inode, data extents, checksum tree	6
Between inode and orphan items	1
Between root tree and file system trees	3
Between root tree and orphan items	1

Table 2: Number of Btrfs rules by datatype

ing process allows each tree to be checked independently. Our online checking approach for btrfs snapshots is much simpler, as described in Section 4.

3.3 Deriving Consistency Invariants

Many consistency properties of the file system, such as “all live data blocks are marked in the block bitmap” are statements about the whole file system, requiring a full disk scan for verification. Instead, we transform each consistency property to a corresponding *consistency invariant*, which is an assertion that must hold for *all* transactions, thereby preserving the consistency property. A bug is detected when any invariant is violated.

We structure a consistency invariant as an implication $A \Rightarrow B$. The premise A always involves an update to data structure fields. When such an update occurs then the conclusion B , which may or may not involve updates to data structure fields, must be true to preserve the invariant. If B does involve updates, then a converse $B \Rightarrow A$ invariant will also exist, and we refer to the two invariants as a $A \Leftrightarrow B$ biconditional invariant.

Below, we use examples to show how consistency properties for various data structures shown in Tables 1 and 2 can be transformed into invariants. Our experience with the ext3 and the btrfs file systems shows that the consistency rules and thus the corresponding invariants for unrelated file systems are very different because they are tightly tied to the implementation. For instance, it is clearly not possible to use the file system checker for one file system to check another. Nonetheless, Section 4 describes the framework we provide to simplify the process of interpreting metadata and checking invariants for different file systems.

3.3.1 Immutable Fields and Range Checks in Ext3

The ext3 fsck program checks for valid values in several fields of the superblock and group descriptor table (rows 1 and 2 in Table 1). Many of these fields are initialized when a file system is created and should never be modified by

a running file system. These properties can be easily verified by marking these fields as immutable and checking them when a superblock or a group descriptor table is updated. Another class of consistency properties relates to simple range checks on the values of given fields. These properties can also be easily verified at each transaction.

3.3.2 Block Bitmap and Block Pointers in Ext3

One of the most important consistency properties in ext3 is that no data block may be doubly allocated, that is, every block pointer (whether it is found in a live inode or indirect block) must be unique or 0 (row 8 in Table 1). The corresponding consistency invariant roughly translates to “if a transaction changes a block pointer to a nonzero value, it must not be used anywhere else”. Checking this invariant would be expensive if we simply scanned all inodes and indirect blocks searching for another instance of the pointer.

A more efficient method would be to keep a bitmap of allocated blocks, similar to the block bitmap maintained by the file system itself. This observation leads to another consistency property of ext3: when a block pointer is set to point at some data block, the corresponding bit in the file system’s bitmap must toggle from unset to set. The biconditional invariants shown below helps prevent double allocation.

block pointer set to N from 0 \Leftrightarrow bit N set in bitmap
 block pointer set to 0 from N \Leftrightarrow bit N unset in bitmap

Checking these invariants requires two versions of a block and the block bitmap, the previous version read from disk, and the version updated by the file system. Block differencing allows detecting updates to the blocks. The previous version is assumed to be correct and can be cached in the Recon metadata cache to improve performance, when it is initially read from disk. The updated block version is untrusted and written to disk only after the invariants above are satisfied.

The invariants above 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 exactly 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. This check (shown below) only requires a scan of the current transaction rather than the whole file system. These consistency invariants together guard against double block allocation.

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

3.3.3 Inode, Inode Bitmap and Orphan List in Ext3

The inode has a deletion time field (dtime) with the constraint that when the link count is non-zero, the deletion time must be zero. Initially, we created the following invariants for an inode based on this property:

link_count set to $> 0 \Leftrightarrow$ dtime set to 0
 link_count set to 0 \Leftrightarrow dtime set to current time $\neq 0$

What complicates matters is that ext3 maintains a persistent “orphan list” that tracks the list of inodes that no longer exist in the filesystem namespace (i.e. the inodes are unlinked from every directory), but cannot yet be deallocated because one or more open file handles exist for that inode. In case of an unclean shutdown, the orphan list is used to free the inodes and data blocks that would otherwise have leaked. Unfortunately, the inodes in the orphan list are linked together by overloading the dtime field, which serves as the pointer to the next element in the orphan list (the first inode in the orphan list is pointed to by a field in the superblock). Thus, when an inode’s link count becomes zero, its dtime should be non-zero, *unless it is the last element in the orphan list*. The last element breaks the usual relationship between link_count and dtime. The invariants are modified to be the following:

link_count set to $> 0 \Leftrightarrow$ dtime set to 0
 link_count set to 0 \Leftrightarrow (dtime set to $!0 \parallel is_orphan_tail$)

A further complication arises because the file system zeroes out an inode (the mode field is set to 0) when an entire inode block is unused. In this case, the link_count and dtime are both zero, but the inode is not the tail of the orphan list. After analyzing this complex behavior, we found that the consistency property that the file system aims to maintain is that *exactly* one of these properties must be true: dtime $\neq 0$, link_count $\neq 0$, mode == 0, and is_orphan_tail(inode). These properties can be checked using two biconditional invariants shown below (row 7 in Table 1).

(link_count set to $> 0 \parallel mode$ set to 0) \Leftrightarrow dtime set to 0
 link_count set to 0 \Leftrightarrow (dtime set to $!0 \parallel is_orphan_tail \parallel mode$ set to 0)

Similar to the block bitmap, the inode bitmap tracks the inode allocation status. The corresponding invariants involve the inode bitmap, link_count, dtime and the orphan_list (not shown).

3.3.4 Directories in Ext3

The inter-directory consistency properties essentially ensure that the directory tree forms a single, bidirected² tree (row 10 in Table 1). This complex consistency property requires two biconditional and two regular invariants as shown below. The notation “Type, ID, Change” below describes a change to a metadata block of type “Type” (directory in this example) with identity ID (we use a directory’s inode number as its identity). The “Change” item indicates the type of change involved and may reference either the old or new value. We discuss this notation in more detail in Section 4. Whenever a directory is linked (or its “.” entry changes), the first biconditional invariant verifies the directory tree’s bidirectional property by checking that the directory’s parent (child) has the directory as its child (parent). This check also verifies that a directory does not have multiple parents, i.e., a directed acyclic graph (DAG) is not present. When a directory is unlinked (or moved), the second invariant checks that it is unlinked on both sides (although not shown, we also check that the directory is empty). When a directory’s “.” entry is updated, the third invariant checks that the “.” entry points to itself.

```
Dir,id=Z,new.dotdot=X ⇔ Dir,id=X,new_entry.inode=Z
Dir,id=Z,old.dotdot=X ⇔ Dir,id=X,rem_entry.inode=Z
Dir,id=Z,new.dot=X   ⇒ X==Z
Dir,id=Z,new.dotdot=X ⇒ is_ancestor(ROOT,Z)
```

Finally, the fourth invariant checks that a directory update does not cause cycles. The bidirectional checks above 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 bidirectional invariants, but it would create a disconnected cycle consisting of a and b. The fourth invariant 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.

3.3.5 Parent and Child Extents in btrfs

There are three consistency properties between the extents associated with *any* btree node and its child node or leaf (row 2 in Table 2). These properties are shown below. The blockptr and bytenr fields are physical byte locations on disk, the generation number is the id of the transaction in which an extent was last updated, and the key helps locate items in a btree. The first two properties make use of self-identifying extents, helping detect lost or misdirected writes [6]. The last property checks the

²A bidirected tree is the directed graph obtained from an undirected tree by replacing each undirected edge by two directed edges in opposite directions.

integrity of the back reference from a child to its parent.

```
foreach ptr i in a parent:
  parent.ptr[i].blockptr == child.header.bytenr
  parent.ptr[i].generation == child.header.generation
  parent.ptr[i].key == child_node.ptr[0].key ||
  child_leaf.items[0].key
```

In the btrfs copy-on-write update model, when a btree leaf or node extent is updated, its parent is updated also. This update chain continues until the root of the tree. We convert the consistency properties above to invariants by checking the properties for only the updated extents and ensuring that the update chain reaches the root of the tree.

3.3.6 Inodes in File System Tree in btrfs

An btrfs inode consistency property is that inode items are unique by objectid (i.e., inode number) in each file-system tree (row 5 in Table 2). Checking such uniqueness requires a full scan of the tree. The corresponding invariant is that when an inode item is added to a file-system tree, it should not exist in that tree. We can avoid traversing the tree for each added inode because btrfs allocates inode numbers in sequence without worrying about holes (deleted inodes) from a 64 bit namespace. We can check that all new inode items have unique inode numbers and they are greater than the largest inode number in the tree. The latter is easy to find in a btree and can be cached.

4 Checking Invariants

Recall that a consistency invariant is structured as an implication $A \Rightarrow B$. The premise A always involves an update to file-system metadata structures, such as updates to an inode field, directory entry, btree items, etc., and the conclusion B must be true to preserve the invariant. Invariants can be implemented efficiently because they are checked only when a transaction updates the fields associated with the premise. The conclusion may or may not involve updates to data structure fields, and it may access the pre-update or post-update file-system state. For example, the first two directory invariants described in Section 3.3.4 access the old and the new values of the parent’s (dotdot) directory entry. Below, we first describe the Recon architecture and then our method for checking invariants.

Figure 1 shows the architecture of the Recon system. Recon uses the Linux device mapper framework to interpose on all file system requests at the block layer, allowing it to track all file-system metadata accessed from disk. Table 3 shows the calls made by the block layer to invoke file-system independent (generic) Recon functionality. On a block read, recon_read caches a metadata block in its read cache. The read cache allows accessing the pre-update file-system state (i.e., disk state) efficiently.

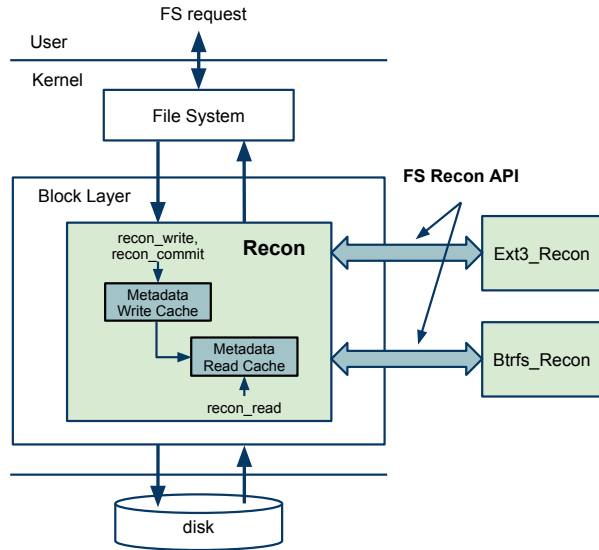


Figure 1: The Recon Architecture

Generic API	
recon_init	initializes recon state
recon_read	caches block in metadata read cache
recon_write	caches block in metadata write cache
recon_commit	invokes file-system specific Recon API

Table 3: Generic Recon API

On a block write, `recon_write` caches the updated block in a metadata write cache, which allows accessing the post-update file-system state. Note that the post-update state is derived by superposing the write cache on the read cache. When a transaction commits, `recon_commit` checks invariants by invoking file-system specific Recon API calls (discussed below). When all applicable invariants are checked successfully, the transaction is allowed to commit, after which the write cache is merged with the read cache, thus updating Recon’s view of file-system state.

The read cache is trusted because its blocks have been verified previously. These cached blocks can be evicted and reread from disk because the disk contents are also trusted. We use a simple LRU mechanism for cache replacement when the cache grows beyond a user-configurable limit. As discussed in Section 5, cache misses are the main source of IO performance overhead for Recon. The write cache may contain corrupt data and thus any code accessing this cache must perform careful validation.

Checking invariants is a three step process. First, we infer the type and identity of data structures in updated metadata blocks in a transaction. Second, these data structures are compared with their previous versions to derive a set of data structure updates that we call change records. Finally, before a transaction is committed to disk, the invariants are checked by using change records and the pre-

and post-transaction file-system state. We describe these steps in more detail below.

4.1 Inferring Type and Identity Information

Table 4 shows the file-system specific Recon API. The `references` function is invoked by `recon_read` to obtain type and identity information for data structures in blocks that are referenced by the newly cached block. For example, when an inode block is read in ext3, this function provides type and identity (e.g., physical location) information for any indirect blocks referenced by inodes in the block. As a result, when an indirect block is read from disk, Recon can easily determine its block type. The ext3 `references` function requires significant machinery because it needs to support all of its metadata block types. The btrfs `references` function is much simpler because metadata extents are either nodes or leaves, and nodes contain most extent pointers. In fact, the btrfs function helps improve type classification performance but is not essential because btrfs extents are self identifying.

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 each updated metadata block to obtain type information for the metadata blocks referenced by the updated block. This function validates the updated blocks by checking that any pointers and size fields within the block are within reasonable bounds so that further processing is not compromised. One interesting case occurs when the type of an updated block is not known, e.g., when a newly allocated directory block appears in the transaction. Recon ignores an unknown block and only processes updated blocks whose types are known. At least one of these blocks (e.g., an inode block) must point to the unknown directory block or else the unknown block would not be reachable in the file system image. At the end of write processing, if an unknown block is found, Recon signals a reachability invariant violation.

4.2 Generating Change Records

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. A change record has the following format:

[type, id, field, oldval, newval]

The type is a data structure (e.g., inode), the id is the identity of a specific object of the given type (e.g. inode nr.), and field is a field in the object (e.g. inode size). The values oldval and newval are the old and new values of the corresponding field. When oldval is ϕ (a sentinel value), the item associated with this change record is newly created or allocated. When newval is ϕ , the item associated

FS Recon API	Invoked by	
references	recon_read	provides type and id information for data structures in referenced blocks
process_write	recon_commit	provides type and id information for data structures in referenced blocks
process_txn	recon_commit	generates change records
check	recon_commit	checks invariants using change records and metadata read/write caches

Table 4: File-system specific Recon API

with this change record was deallocated or destroyed. For ext3, the type value includes direct blocks, indirect blocks, etc. The id value includes block numbers, group descriptor numbers, etc. For btrfs, the change record incorporates the key used for traversing a btree. This key consists of a type, an objectid and a type-specific offset. The type field in the key identifies a btrfs data structure and is used as the type value in the change record. The id value consists of the tree id, identifying the tree in which the key is located, and the objectid and offset fields in the key.

While the process of comparing data structures is clearly file-system specific, we found two common cases. When data structures have fixed size and are located in well defined positions in blocks, 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. Currently, we use simple macros on the data structures to generate these tables. When data structures have variable length or are unsorted, such as directory entries in ext3, or extent items in btrfs, we derive three sets consisting of new items, deleted items and modified items. Change records can be trivially generated from these sets. For example, a new item has the sentinel value for oldval in its change record. Besides these cases, the ext3 code for generating change records must handle block allocation and deallocation. These are detected when an updated block sets or unsets a block pointer. A newly allocated block or a deallocated block is compared with a zeroed block for generating change records.

When processing a btrfs file system tree, we generate three disjoint sets of items: new item set, deleted item set and modified item set. We find these sets by traversing the write cache, beginning at the snapshot root, and building a set of items from the updated blocks in the snapshot tree (“write set”). Then, we traverse the read cache, beginning with the snapshot root in the read cache. We stop traversing a disk pointer in the read cache when it exists in the write set also. As a result, we traverse the parts of the read cache that are no longer referenced from the write cache (“old set”). The old set contains deleted or modified items. Intersecting the write set with the old set provides the new, deleted and modified item sets. These sets are used to generate change records.

Section 3.2 describes the complicated strategy used by the btrfsck program for checking multiple file system trees efficiently. We use a much simpler method and process each updated file system tree separately to ensure its con-

sistency. The reason this is not a serious concern is because our cost is proportional to the number of updated blocks (likely much smaller than the entire file system) and updated trees (likely much smaller than the total number of trees in the system).

4.3 Checking Invariants

Invariant checking consists of pattern matching and a final processing phase. Change records are pattern matched with the premise of an invariant. When such a match occurs, some invariants accumulate bookkeeping information until the end of the transaction and then require some final processing. Invariants can be checked at any time. Below, we show some examples from Section 3.3.

The superblock field immutability checks in Section 3.3.1 are implemented by simply pattern matching a change record of the form [Superblock, *, magic_field, *, *]. Here * matches any value. The existence of this record indicates that the magic field was modified.

Rules that involve relationships between changes to fields require matching multiple patterns. For example, the Ext3 block allocation invariant described in Section 3.3.2 is “block pointer set to N from 0 \Leftrightarrow bit N set in bitmap”. Whenever we see [*, *, block_pointer, 0, X], we insert a flag with key X into a rule-specific bookkeeping table indicating a new pointer has been set to X. When we see [block bitmap, Y, *, 0, 1], we insert a different flag into the same table with key Y indicating bit Y in the allocation bitmap is set. During final processing, we verify that for each key in the table, both flags are set. Otherwise the invariant is violated.

For the btrfs inode uniqueness check in Section 3.3.6, a change record matching [Inode, *, (X, Y, *), ϕ , *] indicates that a new inode has been allocated in tree X with inode number Y. We immediately verify that Y is greater than our cached copy of the largest inode in tree X. We insert key (X,Y) in a bookkeeping table to signify that the inode number Y is used in tree X, unless this key has already been inserted, in which case the invariant is violated. During final processing, our cached value is updated to the max value of the newly allocated inode numbers in each tree.

The right hand sides of some rules require making queries on the post-update file-system state. The last check in Section 3.3.4, “Dir, id=Z, new.dotdot=X \Rightarrow is_ancestor(ROOT, Z)” is used to ensure connectivity from directory Z to the root. We match [Dir, Z, dot_dot,

*, *] and insert Z into a table. When the rule is being checked, we execute `is_ancestor(ROOT, Z)`, which uses the superimposed write/read cache to iteratively traverse the directory hierarchy from Z, until either the root directory is reached or a cycle is detected.

4.4 Handling Invariant Violation

The final problem for an online consistency checker like Recon is dealing with invariant violations. It is extremely important to ensure that recovery from a violation is correct, 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 (note that it is not corrupt but may have lost data), or it can be restarted automatically and transparently to applications [28]. 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 we could continue operating the file system (this may preserve data but the file system could be corrupt). Finally, it may be possible to repair file system data structures dynamically [11], an approach that seems feasible in our system because each transaction typically has a small number of updates.

5 Evaluation

Here, we evaluate our `ext3` implementation of Recon in terms of its coverage of the consistency properties of `ext3`, its ability to detect corruption of `ext3` metadata online, and its performance.

5.1 Completeness and Complexity

We have implemented most of the checks performed by the `e2fsck` file system checker, as encapsulated by the SQCK rules. There are a small number of properties that we do not check, related to the bad blocks file, the lost+found directory, the OS-specific fields in inodes, and features such as the extended attributes/ACLs. 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.

Errors in the Recon code introduce a new threat to normal system operation. Since Recon does not modify file system data, it is unlikely to introduce new corruption, but it might incorrectly flag a transaction as dangerous. Depending on the action taken at that point, such false positives might lead to data loss. We argue that a system checking the file system is easier to make correct than the file system itself. Our entire system is 2948 lines of C. Of

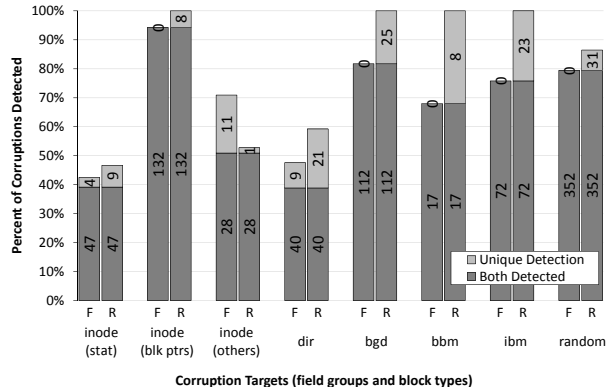


Figure 2: Comparison of corruption detection accuracy

these, 1137 lines are in the generic framework which can be reused across file systems, 1235 are for interpreting the `ext3` metadata, and only 576 lines are involved in checking the invariants. Our dependence on the journal checkpointing code adds another 311 lines. This is considerably smaller than all of `jbd` (3179 loc), `ext3` (8062 loc), and the `vfs` layer above it (28,190 loc). Additionally, the code required to do the checking is simpler for several reasons. 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. The implementation of each invariant check is orthogonal to the other checks - each rule uses its own data structures to keep track of properties that must be verified. 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. The bulk of the complexity lies in the maintenance of the metadata-interpreting structures. We hope to address this by developing a systematic way to describe and interpret these structures.

5.2 Ability to Detect Corruption

An ideal system of evaluation would either inject subtle bugs in the filesystem code or reproduce bugs which had happened in the past, and catch them when they manifest. Bugs that are subtle enough to not be quickly found in a heavily-used filesystem, however, are hard to design. We settled for deliberately injecting corruption of bytes within metadata blocks. This type of corruption could result from several types of bugs (i.e. setting values in arbitrary fields incorrectly) both within the file system and within the operating system overall. 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.

We compare Recon against `e2fsck` by corrupting meta-

data 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 when we begin. 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 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 filesystem 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.

One limitation of our corruption framework is that we only corrupt blocks that the file system is already modifying in some transaction. In particular, we never corrupt superblock fields since the running file system never includes writes to it. We do not consider this a serious limitation since nearly all superblock corruptions would be trivially detected.

For these experiments, we use an 8 GB file mounted as a loop device for our filesystem. This simplified the restoration of the file system following each corruption experiment. Figure 2 summarizes the results of our corruption experiments for Recon and `e2fsck`. We show two bars for each metadata type. The bottom portion of each pair shows the percent of corruptions that were detected by both Recon and `e2fsck`. The top portion shows the percent of corruptions that were only detected by the given 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 is all the block pointer fields, and the third group is everything else. In most cases, our coverage is nearly identical to `e2fsck`. 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. Both checkers are effective at catching corruption of block pointers, however. Recon achieves 100% on these because it checks all inodes, while `e2fsck` ignores ones that are unused. Although the file system consistency is not affected by changes to unused inodes, it is still useful to detect the corruption when it happens, since it is an indicator of a bug somewhere in the system. For the final set of inode fields, `e2fsck` detects several corruptions on extended attributes and os-specific fields that Recon currently does not check.

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 [19]	19.9 GB

Table 5: Benchmark Characteristics

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. As one example, Recon achieves 100% detection for block group descriptor 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, ones. These benefits of Recon also make it more effective at detecting fully random corruption. Again, neither checker can achieve 100% accuracy since some of the corruptions hit fields that are not related to the structural consistency of the file system.

Finally, we observe that the file system still contains errors after the `e2fsck` repairs for 36 of the 824 total corruptions that it detects, giving a detectable failure rate of 4.4%. Two of these failures occurred after corrupting a single byte in a single metadata block. In our experiments, we unmount the file system and check it 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, even if it only detects the same problems as `e2fsck`.

5.3 Performance

All performance tests are done on a 1 TB ext3-formatted file system on a machine with 2GB total RAM and dual 3 GHz Xeon CPUs. In all cases, we report the average of 5 runs; all tests are done with cold caches on a freshly booted system. We use the Linux port of Sun’s FileBench (version 1.4.8.fsl.0.8) with the application emulation workload personalities³. 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[19]⁴. Table 5 summarizes the basic characteristics of our benchmarks. We used default settings for all other parameters. For Fileserver, we reduced the default file size to 32k to increase the ratio of meta-

³The OLTP personality did not work in the version we obtained.

⁴The full profile used in the experiments is available at <http://url.removed.for.blinding>

Setup (seconds)	Cache=64MB, Journal=32MB			Cache=128MB, Journal=64MB			Cache=256MB, Journal=128MB		
	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)

data to data in the file system. We describe the performance of Recon compared to native for both the initial benchmark setup (Table 6), which involves heavy metadata writes, and the actual workload execution (Figure 3). In our current implementation, we cannot evict blocks from our metadata cache that have not yet been checkpointed to the filesystem by the journaling code. Thus, the metadata cache size must be larger than the journal size. We present results for three different cache/journal sizes, showing both native and Recon performance in all cases.

During the benchmark setup, when many files are being created, there is a significant cost to Recon, particularly for small cache sizes. As the cache size increases, however, the impact is rapidly reduced. At a 128MB metadata cache, Recon actually improves performance in some cases, most likely because of the ability to satisfy some file system read requests from the cache. The impact of Recon is less noticeable during normal benchmark operations. With a 64MB metadata cache, 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 3), where performance degrades with an increasing cache size. We believe this is the result of memory pressure, but have not fully investigated at this point. Overall, a 128MB metadata cache with a 64MB journal gives the best results for all workloads, with only 8% degradation. Given the growth in main memory sizes, these seem to be quite modest memory requirements for the reliability benefits that Recon can deliver.

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 4) runtime verification.

6.1 Handling file system bugs

File system bugs can either be detected statically or tolerated at runtime. Bug finding tools, based on model checking [29, 31] and static analysis [23], 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 avail-

able, or easy to deploy in live systems [5]. These limitations can be addressed by tolerating bugs at runtime.

EnvyFS [7] applies N-version programming for tolerating file system bugs. It leverages the common VFS interface to pass each file system request received by the VFS layer to 3 child file systems. The results are then compared and the majority result is delivered back to the user. To help deal with the space overhead of storing all data in 3 file systems, EnvyFS includes 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 since operations must be issued to at least 2 file systems and the results compared before an answer is returned. Also, subtle differences in the semantics of the implementations can make it hard to compare results.

Our approach can be viewed as a variant of N-version programming in which the file system’s consistency invariants are rechecked by Recon independently, but the rest of the file system functionality is not reexecuted. Therefore, Recon can be more efficient, but it cannot reuse an existing file system variant for comparison.

Membrane [28] proposes tolerating bugs by restarting a file system that has failed, transparently to applications. This approach assumes that file system bugs will lead to 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 waiting for the file system to crash, a restart could be initiated when Recon detects an inconsistent transaction.

6.2 Checking file system consistency

SQCK [14] 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 [16]. 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.

Many enterprise storage file systems perform online scan and repair in the background. ZFS provides the abil-

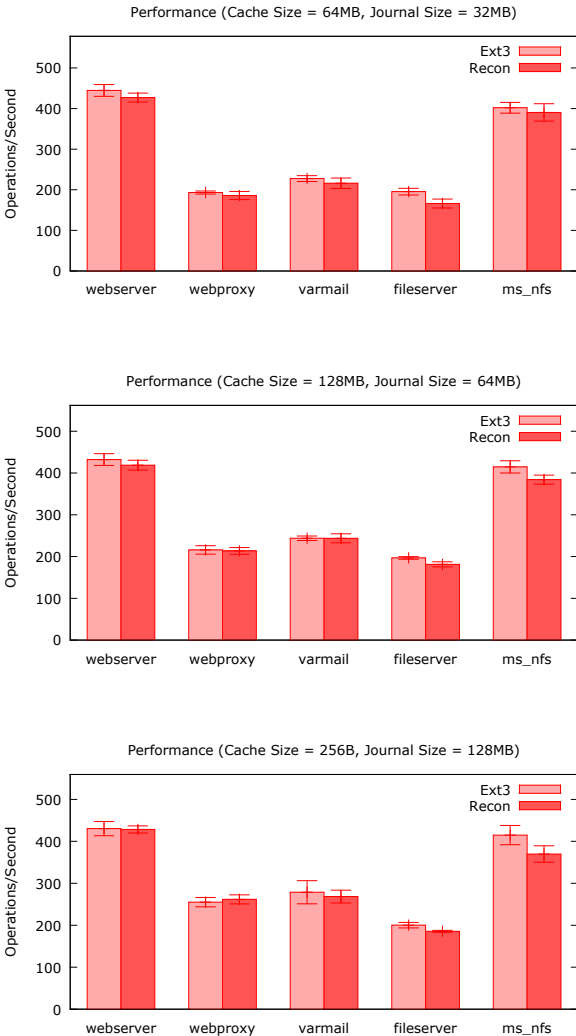


Figure 3: Performance on FileBench workloads for varying metadata cache sizes

ity to scrub disks and repair corrupt blocks that have redundant copies [8]. 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 the online check is time-taking and resource intensive.

6.3 Interpreting file system semantics

Semantically-smart disks use probing to gather detailed knowledge of file system behavior [26]. This knowledge is used below the block interface to transparently improve performance or enhance functionality, such as by implementing track-aligned extents and secure delete. Our work on verifying consistency invariants by inferring file system information builds on several ideas from

semantically-smart disks.

6.4 Runtime verification

The XN storage system of the Xok exokernel is designed to protect library file systems that manage their own disk blocks [18]. XN uses a file-system specific function called `own()`, similar to the references function described in Section 4, that returns for a meta-data block, the blocks controlled by the meta-data block. This function allows XN to verify that a file system can only access blocks that are currently allocated to it. Further, XN can use a file-system specific function called `reboot()` that traverses the entire file-system tree and detects whether the file system is crash consistent. The authors mention that implementing `reboot()` efficiently without a full file-system traversal may be possible, but this task is left as future work. In a sense, this work shows how crash consistency as well as overall file-system consistency functionality can be verified efficiently. There are two other differences between XN and our work. First, Recon assumes the basic block interface (e.g., `read`, `write`), while file systems using XN use an extended block interface (e.g., `allocate`, `read`, `write`, `deallocate`) which allows easier verification. For example, file systems provide block type information to XN while Recon needs to infer this information. Second, XN focuses on protecting file systems from each other and thus 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 and pointer creation/removal [25]. This interface allows enforcing invariants based on pointer relationships, 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. The iSWAT system generates program invariants based on values stored to memory and uses these invariants to detect permanent hardware failures [24]. Program invariants have been used to repair data structures [11] and patch buggy deployed software [20]. 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 [27, 9]. Specifications based on temporal logic are checked continuously during execution by observing input/output behavior, thereby providing assurance that the implementation matches its specification.

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 [2], 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 consistency invariants. As a result, Recon detects random corruption at runtime almost as well as the file system checker. It has low overhead because the data it interprets has likely been accessed or will be accessed by the file system also.

In the future, we plan to explore replacement policies for the metadata cache, that work well with the file system's replacement policy. We also plan to investigate automated methods for learning the consistency invariants, rather than reverse engineering the file system and transaction format.

References

- [1] ext3. <http://en.wikipedia.org/wiki/Ext3>.
- [2] iptables. <http://en.wikipedia.org/wiki/Iptables>.
- [3] NTFS. <http://en.wikipedia.org/wiki/NTFS>.
- [4] XFS. <http://en.wikipedia.org/wiki/XFS>.
- [5] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the EuroSys conference*, pages 187–198, 2009.
- [6] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *Transactions of Storage*, 4(3):1–28, 2008.
- [7] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating file-system mistakes with envyfs. In *Proceedings of the USENIX Technical Conference*, June 2009.
- [8] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.
- [9] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the ACM OOPSLA*, pages 569–588, 2007.
- [10] Oracle Corporation. Btrfs. <http://btrfs.wiki.kernel.org>.
- [11] Brian Demsky and Martin C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering*, 32(12):931–951, 2006.
- [12] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000.
- [13] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007.
- [14] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2008.
- [15] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, November 1987.
- [16] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. 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.
- [17] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Technical Conference*, 1994.
- [18] Frans M. Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazikres, Thomas Pinckney, Robert Grimm, John Jannotti, , and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 52–65, 1997.
- [19] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2010.
- [20] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong,

- Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 87–102, 2009.
- [21] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 802–811, 2005.
- [22] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.
- [23] Rubio-González, Cindy, Gunawi, Haryadi S., Ben Liblit, Arpaci-Dusseau, Remzi H., Arpaci-Dusseau, and Andrea C. Error propagation analysis for file systems. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 270–280, 2009.
- [24] Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 70–79, 2008.
- [25] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2006.
- [26] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies*, pages 73–88, 2003.
- [27] Oleg Sokolsky, Usa Sammapun, Insup Lee, and Je-sung Kim. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science*, 144:91–108, May 2006.
- [28] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2010.
- [29] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2006.
- [30] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 243–257, 2006.
- [31] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.
- [32] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: a ZFS case study. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2010.