Life or Death at Block-Level

Muthian Sivathanu, Lakshmi N. Bairavasundaram,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{*muthian, laksh, dusseau, remzi*}*@cs.wisc.edu*

Abstract

A fundamental piece of information required in intelligent storage systems is the *liveness* of data. We formalize the notion of liveness within storage, and present two classes of techniques for making storage systems liveness-aware. In the *explicit notification* approach, we present robust techniques by which a file system can impart liveness information to storage through a "free block" command. In the *implicit detection* approach, we show that such information can be inferred by the storage system efficiently underneath a range of file systems, without changes to the storage interface. We demonstrate our techniques through a prototype implementation of a *secure deleting disk*. We find that while the explicit interface approach is desirable due to its simplicity, the implicit approach is easy to deploy and enables quick demonstration of new functionality, thus facilitating rapid migration to an explicit interface.

1 Introduction

> "Life is pleasant. Death is peaceful. It's the transition that's troublesome." *Isaac Asimov*

Smarter storage systems need to understand whether blocks are live or dead. Previous work has demonstrated the utility of such knowledge: dead blocks can be used to store rotationally optimal replicas of data [33] or to provide zero-cost writes [31], and failure recovery time can be reduced by restoring only live blocks [23].

Unfortunately, liveness information is not available within modern storage systems, due to the narrow block-based interface between file systems and storage [5, 9]. Storage systems simply observe block-level reads and writes and thus are not aware of logical operations (such as deletes) issued by the file system. This limitation precludes many storage level optimizations [12, 18, 23] and makes others less effective [31, 32, 33].

In this paper, we address this limitation by presenting techniques by which storage systems can be imparted with liveness information. We perform a qualitative and quantitative comparison of two approaches. With *explicit notification*, we augment the interface to storage with a new "free block" command; file systems must be modified to properly use it. With *implicit detection*, we develop techniques to enable the storage system to infer liveness information without any change to the interface.

To better evaluate these approaches, we first formalize the notion of liveness within storage. Specifically, we identify three useful classes of liveness (content, block, and generation liveness), and present techniques for explicit and implicit tracking of each type. Because techniques for imparting liveness information are dependent on the characteristics of the file system, we study a range of file systems, including ext2, ext3 and VFAT; in doing so, we identify key file system properties that impact the feasibility and complexity of such techniques.

To gain more direct experience with liveness-tracking methods, we design, implement, and evaluate a prototype *secure deleting disk* that shreds blocks that have been logically deleted by the file system, making deleted data irrecoverable [12]. We implement secure delete due to its extreme requirements on the type and accuracy of liveness information.

On the surface, both explicit and implicit approaches have their obvious benefits and drawbacks. Explicit notification promises simplicity of implementation but requires broad industry consensus, while implicit detection suggests ease of deployment but at the cost of complexity. Our analysis, however, reveals more complex trade-offs.

We find qualitatively that the explicit approach is less complicated to design and implement. However, while it may appear straightforward to modify file systems to issue "free block" commands, accurate notification in the presence of crashes entails careful integration with file system consistency management schemes, thus noticeably increasing complexity.

We also find that implicit liveness detection is feasible underneath a range of modern file systems; however, some file system behaviors prohibit certain classes of liveness inference. Therefore, we identify the properties that must hold in order to enable or simplify implicit liveness inference. We also propose and implement minor modifications to file systems to conform to those properties.

Finally, we show that implicit liveness detection can be accurate underneath modern asynchronous file systems; our secure delete prototype utilizes implicit liveness to shred blocks that are inferred to be dead. By proving correct operation of implicit secure delete, we demonstrate that implicit liveness can be used in storage applications with extreme correctness requirements. In evaluating the performance of implicit liveness tracking, we find that it is comparable to the explicit approach.

We conclude that storage systems can more easily implement the explicit approach, if the interface is embellished to support it. However, we see the implicit approach as a complementary instead of competitive technology; because industry consensus on interface change is at best slow-moving, implicit techniques (even if complex) can be of use. Specifically, by deploying a particular technology without explicit interface change, implicit techniques can readily demonstrate possible benefits and thus move industry rapidly towards an explicit change.

The paper is organized as follows. We first present an extended motivation (§2), followed by a taxonomy of liveness (§3), and a list of file system properties that impact techniques for imparting liveness information (§4). We proceed by discussing explicit notification (§5) and implicit detection (§6), and then describe secure deletion (§7). We then describe our initial experience with implicit detection under NTFS, a closed-source file system (§8). Finally, we present a discussion on the relative merits of the implicit and explicit approaches (§9), and finish by discussing related work (§10) and concluding (§11). Appendix A includes a proof of correctness for implicit secure delete.

2 Extended Motivation

In this section, we first present examples of functionality enabled by liveness information, and then motivate two alternative approaches for gathering such information.

2.1 Utility of liveness

Liveness information enables a variety of functionality and performance enhancements within the storage system. Most of these enhancements cannot be implemented at higher layers because they require low-level control available only within the storage system.

Eager writing: Workloads that are write-intensive can run faster if the storage system is capable of *eager writing*, *i.e.*, writing to "some free block closest to the disk arm" instead of the traditional in-place write [8, 31]. However, in order to select the closest block, the storage system needs information on which blocks are live. Existing proposals function well as long as there exist blocks that were never written to; once the file system writes to a block, the storage system cannot identify subsequent death of the block as a result of a delete. A disk empowered with liveness information can be more effective at eager writing.

Adaptive RAID: Information on block liveness within the storage system can also facilitate dynamic, adaptive RAID schemes such as those in the HP AutoRAID system [32]; AutoRAID utilizes free space to store data in RAID-1 layout, and migrates data to RAID-5 when it runs short of free space. Knowledge of block death can make such schemes more effective.

Optimized layout: Techniques to optimize on-disk layout transparently within the storage system have been well explored. Adaptive reorganization of blocks within the disk [21] and replication of blocks in rotationally optimal locations [33] are two examples. Knowing which blocks are free can greatly facilitate such techniques; live blocks can be collocated together to minimize seeks, or the "free" space corresponding to dead blocks can be used to hold rotational replicas.

Smarter NVRAM caching: Buffering writes in NVRAM is a common optimization in storage systems. For synchronous write workloads that do not benefit much from in-memory delayed writes within the file system, NVRAM buffering improves performance by absorbing multiple overwrites to a block. However, in delete-intensive workloads, unnecessary disk writes can still occur; in the absence of liveness information, deleted blocks occupy space in NVRAM and need to be written to disk when the NVRAM fills up. From real file system traces [20], we found that up to 25% of writes are deleted *after* the typical delayed write interval of 30 seconds, and thus will be unnecessarily written to disk. Knowledge about block death within storage removes this overhead.

Intelligent prefetching: Modern disks perform aggressive prefetching; when a block is read, the entire track in which the block resides is often prefetched [22], and cached in the internal disk cache. In an aged (and thus, fragmented) file system, only a subset of blocks within a track may be live, and thus, caching the whole track may result in suboptimal cache space utilization. Although reading in the whole track is still efficient for disk I/O, knowledge about liveness can enable the disk to selectively cache only those blocks that are live.

Faster recovery: Liveness information enables faster recovery in storage arrays. A storage system can reduce reconstruction time during disk failure by only reconstructing blocks that are live within the file system [23].

Self-securing storage: Liveness information in storage can help build intelligent security functionality in storage systems. For example, a storage level intrusion detection system (IDS) provides another perimeter of security by monitoring traffic, looking for suspicious access patterns such as deletes or truncates of log files [18]; detecting these patterns requires liveness information.

Secure delete: The ability to delete data in a manner that makes recovery impossible is an important component of data security [3, 12, 14]. Government regulations require strong guarantees on sensitive data being "forgotten", and such requirements are expected to become more widespread in both government and industry in the near future [1]. Secure deletion requires low-level control on block placement that is available only within the storage system; implementing storage level secure delete requires liveness information within the storage system. We explore secure deletion further in Section 7.

2.2 Acquiring liveness information

Despite the clear benefits of liveness information in storage systems, such information is not currently available. A natural question that arises is how to convey liveness information to storage systems. We discuss two approaches: *explicit notification* and *implicit detection*.

2.2.1 Explicit notification

Explicit notification involves augmenting the existing storage interface with new "allocate block" and "free block" commands, and then modifying file systems to use these commands to explicitly convey liveness information to the storage system. The main benefit of the explicit approach is its potential simplicity; once the new interface is deployed, conveying liveness information is seemingly straightforward.

However, while appearing to be a natural way to achieve our goal, there are a few problems with this approach. First, changing the interface to storage raises legacy issues and requires broad industry consensus. Second, a demand for such a new interface often requires agreement on the clear benefits of the interface, which is difficult to achieve without deployment of the interface - a chicken-and-egg problem.

2.2.2 Implicit detection

Implicit detection is intended to solve the bootstrapping problem in explicit interface evolution. In this approach, the storage system monitors block-level reads and writes issued by the file system from underneath an unmodified interface and infers liveness information implicitly, ideally with no change to the file system above. The implicit approach thus enables demonstration of benefits due to a proposed interface change, thereby making it an evolutionary step towards an eventual interface modification.

Previous work on *semantically-smart* storage systems [2, 23, 24] has explored implicit detection of various forms of file system information from within the storage system, for various storage-level enhancements. The degree of accuracy required from the implicit detection techniques in each case depends on the nature of the application using that information. In X-RAY [2], the storage system utilizes implicit information on file accesses to implement an exclusive storage array cache; inaccurate information in X-RAY simply reduces the potential performance gain. In D-GRAID [23], the storage system utilizes implicit information on the file a block belongs to, in order to place blocks in a fault-isolated fashion, thus improving the availability of the storage system under multiple disk failures; inaccurate information in D-GRAID leads to poor fault isolation, but does not impact correctness because the array still exhibits strictly better availability than traditional RAID. In this paper, we investigate the limits of implicit detection, by considering applications that utilize implicit liveness information in a way that directly impacts correctness.

Liveness type	**Description**	**Currently possible?**	**Example utility**
Content	Data within block	Yes	Versioning
Block	Whether a block holds valid data currently	No	Eager write, fast recovery
Generation	Block's lifetime in the context of a file	No	Secure delete, storage IDS

Table 1: **Forms of liveness.**

The primary concern with *implicit interface evolution* is that it ties the interacting layers together. For example, if the file system changes, the storage system will likely need to change as well. However, this issue may not be as problematic as it seems. On-disk formats evolve slowly, for reasons of backwards compatibility. For example, the Linux ext2 file system, introduced in roughly 1994, has had the same layout for its lifetime. Further, the ext3 journaling file system [29] is backwards compatible with the on-disk layout of ext2 and the new extensions to the FreeBSD file system [6] are backwards compatible as well. We also have evidence that commercial storage vendors are already willing to maintain and support software specific to a file system; for example, the EMC Symmetrix storage system [7] comes with software that can understand most common file systems. These trends point to the commercial viability of an implicit detection approach.

3 Liveness in Storage: A Taxonomy

Having discussed the utility of liveness information within a storage system, we now present a taxonomy of the forms of liveness information that are relevant to storage. Such liveness information can be classified along three dimensions: *granularity*, *accuracy*, and *timeliness*.

3.1 Granularity of liveness

Depending on the specific storage-level enhancement that utilizes liveness information, the logical unit of liveness to be tracked can vary. We identify three granularities at which liveness information is meaningful and useful: content, block and generation. A summary is presented in Table 1.

3.1.1 Content liveness

Content liveness is the simplest form of liveness. The unit of liveness is the actual data in the context of a given block; thus, "death" at this granularity occurs on every overwrite of a block. When a block is overwritten with new data, the storage system can infer that the old contents are dead. An approximate form of content liveness is readily available in existing storage systems, and has been explored in previous work; for example, Wang *et al.*'s virtual log disk frees the past location of a block when the block is overwritten with new contents [31]. Tracking liveness at this granularity is also useful in on-disk versioning, as seen in self-securing storage systems [28]. However, to be completely accurate, the storage system also needs to know when a block is freed within the file system, since the contents stored in that block are dead even without it being overwritten.

3.1.2 Block liveness

Block liveness tracks whether a given disk block currently contains valid data, *i.e.*, data that is accessible through the file system. The unit of interest in this case is the "container" instead of the "contents". Block liveness is the granularity required for many applications such as intelligent caching, prefetching, and eager writing. For example, in deciding whether to propagate a block from NVRAM to disk, the storage system just needs to know whether the block is live at this granularity. This form of liveness information cannot be tracked in traditional storage systems because the storage system is unaware of which blocks the file system thinks are live. However, a weak form of this liveness can be tracked; a block that was never written to can be inferred to be dead.

3.1.3 Generation liveness

The generation of a disk block is the lifetime of the block in the context of a certain file. Thus, by death of a generation, we mean that a block that was written to disk (at least once) in the context of a certain file becomes either free or is reallocated to a different file. Tracking generation liveness ensures that the disk can detect every logical file system delete of a block whose contents had reached disk in the context of the deleted file. An example of a storage level functionality that requires generation liveness is secure delete, since it needs to track not just whether a block is live, but also whether it contained data that belonged to a file generation that is no longer alive. Another application that requires generation liveness information is storage-based intrusion detection. Generation liveness cannot be tracked in existing storage systems.

3.2 Accuracy of liveness information

The second dimension of liveness is accuracy, by which we refer to the degree of trust the disk can place in the liveness information available to it. Inaccuracy in liveness information can lead the disk into either overestimating or underestimating the set of live entities (blocks or generations). The degree of accuracy required varies with the specific storage application. For example, in delete-squashing NVRAM, it is acceptable for the storage system to slightly overestimate the set of live blocks, since it is only a performance issue and not a correctness issue; on the other hand, underestimating the set of live blocks is catastrophic since the disk would lose valid data. Similarly, in generation liveness detection for secure delete, it is acceptable to miss certain intermediate generation deaths of a block as long as the latest generation death of the block is known.

3.3 Timeliness of information

The third and final axis of liveness is timeliness, which defines the time between a death occurring within the file system and the disk learning of the death. In the explicit notification approach, if the file system delays "free" notifications (similar to delayed writes), there will be a time lag before the disk learns of a block or generation death. Similarly, in the implicit approach, the periodicity with which the file system writes metadata blocks imposes a bound on the timeliness of the liveness information inferred. In many applications, such as eager writing and delete-aware caching, this delayed knowledge of liveness is acceptable, as long as the information has not changed in the meantime. However, in certain applications such as secure delete, timely detection may provide stronger guarantees.

4 File System Properties

Both explicit and implicit methods for imparting liveness information to storage are dependent on the characteristics of the file system using the storage system. We therefore study the range of techniques required for such liveness notification (or detection) by experimenting underneath three different file systems: ext2, ext3, and VFAT. We have also experimented with NTFS, but only on a limited scale due to lack of source code access; our NTFS experience is described in Section 8. Given that ext2 has two modes of operation (synchronous and asynchronous modes) and ext3 has three modes (writeback, ordered, and data journaling modes), all with different update behaviors, we believe these form a rich set of file systems.

We first begin with a brief background on the various file systems and then outline some high level behavioral properties of a file system that are relevant in the context of liveness information. In the next two sections, we discuss how these properties influence different techniques for storage-level liveness tracking.

4.1 File system background

In this subsection, we provide some background information on the various file systems we study. We discuss both key on-disk data structures and the update behavior.

4.1.1 Common properties

We begin with some properties common to all the file systems we consider, from the viewpoint of liveness tracking. At a basic level, all file systems track at least two kinds of on-disk metadata: a structure that tracks allocation of blocks (*e.g.*, bitmap, freelist), and index structures (*e.g.*, inodes) that map each logical file to groups of blocks.

A common aspect of the update behavior of all modern file systems is *asynchrony*. When a data or metadata block is updated, the contents of the block are not immediately flushed to disk, but instead, buffered in memory for a certain interval (*i.e.*, the *delayed write interval*). Blocks that have been "dirty" longer than the delayed write interval are periodically flushed to disk. The order in which such delayed writes are committed can be potentially arbitrary, although certain file systems enforce ordering constraints [10].

4.1.2 Linux ext2

The ext2 file system is an intellectual descendant of the Berkeley Fast File System (FFS) [16]. The disk is split into a set of *block groups*, akin to cylinder groups in FFS, each of which contains inode and data blocks. The allocation status (live or dead) of data blocks is tracked through *bitmap blocks*. Most information about a file, including size and block pointers, is found in the file's inode. To accommodate large files, a few pointers in the inode point to *indirect blocks*, which in turn contain block pointers.

While committing delayed writes, ext2 enforces no ordering whatsoever; crash recovery therefore requires running a tool like *fsck* to restore metadata integrity (data inconsistency may still persist). Ext2 also has a synchronous mode of operation where metadata updates are synchronously flushed to disk, similar to early FFS [16].

4.1.3 Linux ext3

The ext3 file system is a journaling file system that evolved from ext2, and uses the same basic on-disk structures. Ext3 ensures metadata consistency by write-ahead logging of metadata updates, thus avoiding the need to perform an fsck-like scan after a crash. Ext3 employs a coarse-grained model of transactions; all operations performed during a certain *epoch* are grouped into a single transaction. When ext3 decides to commit the transaction, it takes an in-memory copy-on-write snapshot of dirty metadata blocks that belonged to that transaction; subsequent updates to any of those metadata blocks result in a new in-memory copy.

Ext3 supports three modes of operation. In *ordered data* mode, ext3 ensures that before a transaction commits, all data blocks dirtied in that transaction are written to disk. In *data journaling* mode, ext3 journals data blocks together with metadata. Both these modes ensure data integrity after a crash. The third mode, *data writeback*, does not order data writes; data integrity is not guaranteed in this mode.

4.1.4 VFAT

The VFAT file system descends from the world of PC operating systems. In this paper, we consider the Linux implementation of VFAT. VFAT operations are centered around the *file allocation table (FAT)*, which contains an entry for each allocatable block in the file system. These entries are used to locate the blocks of a file, in a linked-list fashion. For example, if a file's first block is at address b, one can look in entry b of the FAT to find the next block of the file, and so forth. An entry can also hold an end-of-file marker or a setting that indicates the block is free. Unlike UNIX file systems, where most information about a file is found in its inode, a VFAT file system spreads this information across the FAT itself and the directory entries; the FAT is used to track which blocks belong to the file, whereas the directory entry contains information like size, type information and a pointer to the start block of the file. Similar to ext2, VFAT does not preserve any ordering in its delayed updates.

Property	**Ext2**	**Ext2+sync**	**VFAT**	**Ext3-wb**	**Ext3-ord**	**Ext3-data**
Reuse ordering		×		×	×	×
Block exclusivity	×	×	×			
Generation marking	×	×		×	×	×
Delete suppression	×	×	×	×	×	×
Consistent metadata				×	×	×
Data-metadata coupling						×

Table 2: **File system properties.** *The table summarizes the various properties exhibited by each of the file systems we study.*

4.2 Properties

The update behavior of the file system has a direct influence on the techniques through which liveness information can be imparted to the storage system. Based on our experience with the aforementioned file systems, we identify high-level file system properties that are relevant to liveness tracking. Table 2 summarizes these properties.

Reuse ordering: If the file system guarantees that it will not reuse disk blocks until the freed status of the block (*e.g.*, bitmaps or other metadata that pointed to the block) reaches disk, the file system exhibits *reuse ordering*. This property is necessary (but not sufficient) to ensure data integrity; in the absence of this property, a file could end up with partial contents from some other deleted file after a crash, even in a journaling file system. While VFAT and the asynchronous mode of ext2 do not have reuse ordering, all three modes of ext3, and ext2 in synchronous mode, exhibit reuse ordering.

Block exclusivity: *Block exclusivity* requires that for every disk block, there is at most one dirty copy of the block in the file system cache. It also requires that the file system employ adequate locking to prevent any update to the in-memory copy while the dirty copy is being written to disk. This property holds for certain file systems such as ext2 and VFAT. However, ext3 does not conform to this property. Because of its snapshot-based journaling, there can be two dirty copies of the same metadata block, one for the "previous" transaction being committed and the other for the current transaction.

Generation marking: The *generation marking* property requires that the file system track reuse of file pointer objects (*e.g.*, inodes) with version numbers. Both the ext2 and ext3 file systems conform to this property; when an inode is deleted and reused for a different file, the version number of the inode is incremented. VFAT does not exhibit this property.

Delete suppression: A basic optimization found in most file systems is to suppress writes of deleted blocks. All file systems we discuss obey this property for data blocks. VFAT does not obey this property for directory blocks.

Consistent metadata: This property indicates whether the file system conveys a consistent metadata state to the storage system. All journaling file systems exhibit the consistent metadata property; transaction boundaries in their on-disk log implicitly convey this information. Ext2 and VFAT do not exhibit this property.

Data-metadata coupling: *Data-metadata coupling* builds on the consistent metadata property, and it requires the notion of consistency to be extended also to data blocks. In other words, a file system conforming to this property conveys a consistent metadata state together with the set of data blocks that were dirtied in the context of that transaction. Among the file systems we consider, only ext3 in data journaling mode conforms to this property.

5 Explicit Liveness Notification

We now proceed to the techniques for imparting various forms of liveness information to storage systems. In this section, we discuss the *explicit notification* approach, where we assume that special `allocate` and `free` commands are added to SCSI. As an optimization, we obviate the need for an explicit `allocate` command by treating a `write` to a previously freed block as an implicit `allocate`. Although modifying file systems to use this interface may seem trivial, we find that supporting the `free` command has ramifications in the consistency management of the file system under crashes.

We have modified the Linux ext2 and ext3 file systems to use this `free` command to communicate liveness information; we discuss the issues therein. The `free` command is implemented as an *ioctl* to a pseudo-device driver, which serves as our enhanced disk prototype.

5.1 Granularity of `free` notification

One issue that arises with explicit notification is the exact semantics of the `free` command, given the various granularities of liveness outlined in Section 3. For example, if only block liveness or content liveness needs to be tracked, the file system can be lazy about initiating `free` commands (thus suppressing `free` to blocks that are subsequently reused). For generation liveness, the file system needs to notify the disk of every delete of a block whose contents reached disk in the context of the deleted file. However, given multiple intermediate layers of buffering, the file system may not know exactly whether the contents of a block reached disk in the context of a certain file.

To simplify file system implementation, the file system should not be concerned about what form of liveness a particular disk functionality requires. In our approach, the file system invokes the `free` command for every logical delete. On receiving a `free` command for a block, the disk marks the block dead in its internal allocation structure (*e.g.*, a bitmap), and on a `write`, it marks the corresponding block live. The responsibility for mapping these `free` commands to the appropriate form of liveness information lies with the disk. For example, if the disk needs to track generation deaths, it will only be interested in a `free` command to a block that it thinks is live (as indicated by its internal bitmaps); a redundant `free` to a block that is already free within the disk (which happens if the block is deleted before being written to disk) will not be viewed as a generation death. For correct operation, the file system should guarantee that it will not write a block to disk without a prior allocation; if the `write` itself is treated as an implicit `allocate`, this guarantee is the same as the *delete suppression* property. A `write` to a freed block without an allocation will result in incorrect conclusion of generation liveness within the disk. Note that after a `free` is issued for a block, the disk can safely use that block, possibly erasing its contents.

5.2 Timeliness of `free` notification

Another important issue that arises in explicit notification of a `free` is *when* the file system issues the notification. One option is *immediate notification*, where the file system issues a "free" immediately when a block gets deleted in memory. Unfortunately, this solution can result in loss of data integrity in certain crash scenarios. For example, if a crash occurs immediately after the `free` notification for a block B but before the metadata indicating the corresponding delete reaches disk, the disk considers block B as dead, while upon recovery the file system views block B as live since the delete never reached disk. Since a live file now contains a freed block, this scenario is a violation of data integrity. While such violations are acceptable in file systems such as ext2 which already have weak data integrity guarantees, file systems that preserve data integrity (such as ext3) need to *delay* notification until the effect of the delete reaches disk.

Delayed notification requires the file system to conform to the *reuse ordering* property; otherwise, if the block is reused (and becomes live within the file system) before the effect of the previous delete reaches disk, the delayed `free` command would need to be suppressed, which means the disk would miss a generation death.

5.3 Orphan allocations

Finally, explicit notification needs to handle the case of *orphan allocations*, where the file system considers a block dead while the disk considers it live. Assume that a block is newly allocated to a file and is written to disk in the context of that file. If a crash occurs at this point (but before the metadata indicating the allocation is written to disk), the disk would assume that the block is live, but on restart, the file system views the block as dead. Since the on-disk contents of the block belong to a file that is no longer extant in the file system, the block has suffered a generation death, but the disk does not know of this. The `free` notification mechanism should enable accurate tracking of liveness despite orphan allocations. Handling orphan allocations is file system specific, as we describe below.

5.4 Explicit notification in ext2

As mentioned above, because ext2 does not provide data integrity guarantees on a crash, the notification of deletes can be immediate; thus ext2 invokes the `free` command synchronously whenever a block is freed in memory. Dealing with orphan allocations in ext2 requires a relatively simple but expensive operation; upon recovery, the fsck utility conservatively issues `free` notifications to every block that is currently dead within the file system.

5.5 Explicit notification in ext3

Because ext3 guarantees data integrity in its ordered and data journaling modes, `free` notification in ext3 has to be delayed until the effect of the corresponding delete reaches disk. In other words, the notification has to be delayed until the transaction that performed the delete commits. Therefore, we record an in-memory list of blocks that were deleted as part of a transaction, and issue `free` notifications for all those blocks when the transaction commits. Since ext3 already conforms to the reuse ordering property, such delayed notification is feasible.

However, a crash could occur during the invocation of the `free` commands (*i.e.*, immediately after the commit of the transaction); therefore, these `free` operations should be redo-able on recovery. For this purpose, we also log special `free` records in the journal which are then replayed on recovery, as part of the delete transaction.

During recovery, since there can be multiple committed transactions which will need to be propagated to their on-disk locations, a block deleted in a transaction could have been reallocated in a subsequent committed transaction. Thus, we cannot replay all logged `free` commands. Given our guarantee of completing all `free` commands for a transaction before committing the next transaction, we should only replay `free` commands for the last successfully committed transaction in the log (and not for any earlier committed transactions that are replayed).

To deal with orphan allocations, we log block numbers of data blocks that are about to be written, before they are actually written to disk. On recovery, ext3 can issue `free` commands to the set of orphan data blocks that were part of the uncommitted transaction.

6 Implicit Liveness Detection

In this section, we analyze various issues in implicit detection of liveness from within the storage system. Implicit liveness inference requires the storage system to have semantic understanding [24] of the on-disk format of the file system running above, coupled with careful observation of file system traffic. Because implicit liveness detection is file system dependent, we discuss the feasibility and generality of implicit liveness detection by considering three different file systems: ext2, ext3, and VFAT. In Section 8, we discuss our initial experience with implicit detection underneath the Windows NTFS file system.

Among the different forms of liveness we address, we only consider the granularity and accuracy axes mentioned in Section 3. Along the accuracy axis, we consider *accurate* and *approximate* inferences; the *approximate* instance refers to a strict *over-estimate* of the set of live entities. On the timeliness axis, we address the more common (and complex) case of lack of timely information; under most modern file systems that delay metadata updates, timeliness is not guaranteed. With guarantees of timeliness (*e.g.*, under a synchronously mounted file system), implicit inference of liveness is trivial [24].

6.1 Content liveness

As discussed in Section 3, when the disk observes a write of new contents to a live data block, it can infer that the previous contents stored in that block has suffered a content death. However, to be completely accurate, content liveness inference requires information on block liveness.

6.2 Block liveness

Block liveness information enables a storage system to know whether a given block contains valid data at any given time. To track block liveness, the storage system monitors updates to structures tracking allocation. In ext2 and ext3, there are specific data bitmap blocks which convey this information; in VFAT this information is embedded in the FAT itself, as each entry in the FAT indicates whether or not the corresponding block is free. Thus, when the file system writes an allocation structure, the storage system examines each entry and concludes that the relevant block is either dead or live.

Because allocation bitmaps are buffered in the file system and written out periodically, the liveness information that the storage system has is often stale, and does not account for new allocations (or deletes) that occurred during the interval. Table 3 depicts a time line of operations which leads to an incorrect inference by the storage system. The bitmap block M_B tracking the liveness of B is written in the first step indicating B is dead. Subsequently, B is allocated to a new file I_1 and written to disk while M_B (now indicating B as live) is still buffered in memory. At this point, the disk wrongly believes that B is dead while the on-disk contents of B are actually valid.

Operation	In-memory	On-disk
Initial	$M_B \Rightarrow B$ free	
M_B write to disk		B free
I_1 alloc	$I_1 \rightarrow B$	
	$M_B \Rightarrow B$ alloc	
B write to disk		B written
Liveness belief	B live	B free

Table 3: **Naive block liveness detection.** *The table depicts a time line of events that leads to an incorrect liveness inference. This problem is solved by the shadow bitmap technique.*

To address this inaccuracy, the disk tracks a *shadow copy* of the bitmaps internally [23]; whenever the file system writes a bitmap block, the disk updates its shadow copy with the copy written. In addition, whenever a data block is written to disk, the disk pro-actively sets the corresponding bit in its shadow bitmap copy to indicate that the block is live. In the above example, the write of B leads the disk to believe that B is live, thus preventing the incorrect conclusion from being drawn.

6.2.1 File system properties for block liveness

The shadow bitmap technique tracks block liveness accurately only underneath file systems that obey either the block exclusivity or data-metadata coupling property.

Block exclusivity guarantees that when a bitmap block is written, it reflects the current liveness state of the relevant blocks. If the file system tracks multiple snapshots of the bitmap block (*e.g.*, ext3), it could write an old version of a bitmap block M_B (indicating B is dead) after a subsequent allocation and write of B. The disk would thus wrongly infer that B is dead while in fact the on-disk contents of B are valid, since it belongs to a newer snapshot; such uncertainty complicates block liveness inference.

If the file system does not exhibit block exclusivity, block liveness tracking requires the file system to exhibit data-metadata coupling, *i.e.*, to group metadata blocks (*e.g.*, bitmaps) with the actual data block contents in a single consistent group; file systems typically enforce such consistent groups through transactions. By observing transaction boundaries, the disk can then reacquire the temporal information that was lost due to lack of block exclusivity. For example, in ext3 data journaling mode, a transaction would contain the newly allocated data blocks together with the bitmap blocks indicating the allocation as part of one consistent group. Thus, at the commit point, the disk conclusively infers liveness state from the state of the bitmap blocks in that transaction. Since data writes to the actual in-place locations occur only after the corresponding transaction commits, the disk is guaranteed that until the next transaction commit, all blocks marked dead in the previous transaction will remain dead. In the absence of data-metadata coupling, a newly allocated data block could reach its in-place location before the corresponding transaction commits, and thus will become live in the disk before the disk detects it.

Operation	In-memory	On-disk
Initial	$M_B \Rightarrow B$ alloc	B live
	$I_1 \rightarrow B$	$I_1 \rightarrow B$
B write to disk		B written
I_1 delete	$M_B \Rightarrow B$ free	
I_2 alloc	$I_2 \rightarrow B$	
	$M_B \Rightarrow B$ alloc	
M_B write to disk		B live
Liveness belief		(Missed gen. death)

Table 4: **Missed generation death under block liveness.** *The table shows a scenario to illustrate that simply tracking block liveness is insufficient to track generation deaths.*

For accuracy, block liveness also requires the file system to conform to the delete suppression property; if delete suppression does not hold, a write of a block does not imply that the file system views the block as live, and thus the shadow bitmap technique will overestimate the set of live blocks until the next bitmap write. From Table 2, ext2, VFAT, and ext3 in data journaling mode thus readily facilitate block liveness detection.

6.3 Generation liveness

Generation liveness is a stronger form of liveness than block liveness, and hence builds upon the same shadow bitmap technique. With generation liveness, the goal is to find, for each on-disk block, whether a particular “generation” of data (*e.g.*, that corresponding to a particular file) stored in that block is dead. Thus, block liveness is a special case of generation liveness; a block is dead if the latest generation that was stored in it is dead. Conversely, block liveness information is not sufficient to detect generation liveness because a block currently live could have stored a dead generation in the past. Table 4 depicts this case. Block B initially stores a generation of inode I_1, and the disk thinks that block B is live. I_1 is then deleted, freeing up B, and B is immediately reallocated to a different file I_2. When M_B is written the next time, B continues to be marked live. Thus, the disk missed the generation death of B that occurred between these two bitmap writes.

6.3.1 Generation liveness under reuse ordering

Although tracking generation liveness is in general more challenging, a file system that follows the reuse ordering property makes it simple to track. With reuse ordering, before a block is reused in a different file, the deleted status of the block reaches disk. In the above example, before B is reused in I_2, the bitmap block M_B will be written, and thus the disk can detect that B is dead. In the presence of reuse ordering, tracking block liveness accurately implies accurate tracking of generation liveness. File systems such as ext3 that conform to reuse ordering, thus facilitate *accurate* tracking of generation liveness.

6.3.2 Generation liveness without reuse ordering

Underneath file systems such as ext2 or VFAT that do not exhibit the reuse ordering property, tracking generation liveness requires the disk to look for more detailed information. Specifically, the disk needs to monitor writes to metadata objects that link blocks together into a single logical file (such as the inode and indirect blocks in ext2, the directory and FAT entries in VFAT). The disk needs to explicitly track the "generation" a block belongs to. For example, when an inode is written, the disk records that the block pointers belong to the specific inode.

With this extra knowledge about the file to which each block belongs, the disk can identify generation deaths by looking for *changes in ownership*. For example, in Table 4, if the disk tracked that B belongs to I_1, then eventually when I_2 is written, the disk will observe a change of ownership, because I_2 owns a block that I_1 owned in the past; the disk can thus conclude that a generation death must have occurred in between.

A further complication arises when instead of being reused in I_2, B is reused again in I_1, now representing a new file. Again, since B now belongs to a new generation of I_1, this scenario has to be detected as a generation death, but the ownership change monitor would miss it. To detect this case, we require the file system to track reuse of inodes (*i.e.*, the generation marking property). Ext2 already maintains such a version number, and thus enables detection of these cases of generation deaths. With version numbers, the disk now tracks for each block the "generation" it belonged to (the generation number is a combination of the inode number and the version number). When the disk then observes an inode written with an incremented version number, it concludes that all blocks that belonged to the previous version of the inode should have incurred a generation death. We call this technique *generation change monitoring*.

Finally, it is pertinent to note that the generation liveness detection through generation change monitoring is only *approximate*. Let us assume that the disk observes that block B belongs to generation G_1, and at a later time observes that B belongs to a different generation G_2. Through generation change monitoring, the disk can conclude that there was a generation death of B that occurred in between. However, the disk cannot know exactly *how many* generation deaths occurred in the relevant period. For example, after being freed from G_1, B could have been allocated to G_3, freed from G_3 and then reallocated to G_2, but the disk never saw G_3 owning B due to delayed write of G_3. However, as we show in our case study, this weaker form of generation liveness is still quite useful.

A summary of the file system properties required for various forms of implicit liveness inference is presented in Table 5.

7 Case Study: Secure Delete

To demonstrate our techniques for imparting liveness to storage, we present the design, implementation, and evaluation of a *secure deleting disk* under both explicit and implicit approaches. We first describe implicit secure delete in detail, and then briefly discuss explicit secure delete.

Liveness type	**Properties**
Block$_{Approx}$	Block exclusivity *or* Data-metadata coupling
Block$_{Accurate}$	[Block$_{Approx}$] + Delete suppression
Generation$_{Approx}$	[Block$_{Approx}$] + Generation marking
Generation$_{Accurate}$	[Block$_{Accurate}$] + Reuse ordering

Table 5: **FS properties for implicit liveness detection.** *$Approx$ indicates the set of live entities is over-estimated.*

There are two primary reasons why we chose secure deletion as our case study. First, secure delete requires tracking of generation liveness, which is the most challenging to track. Second, secure delete uses the liveness information in a context where correctness is paramount. A false positive in detecting a delete would lead to irrevocable deletion of valid data, while a false negative would result in the long-term recoverability of deleted data (a violation of secure deletion guarantees). Compared to previous work [24] which functioned only under a simplistic assumption of a synchronously mounted file system, we demonstrate that accurate inference of liveness is feasible underneath a variety of modern file system behaviors.

Our implicit secure deletion prototype is called FADED (A File-Aware Data-Erasing Disk); FADED works underneath three different file systems: ext2, VFAT, and ext3. Because of its complete lack of ordering guarantees, ext2 presented the most challenges. Specifically, since ext2 does not have the reuse ordering property, detecting generation liveness requires tracking generation information within the disk, as described in Section 6.3. We therefore mainly focus on the implementation of FADED underneath ext2, and finally discuss some key differences in our implementation for other file systems.

7.1 Goals of FADED

The desired behavior of FADED is as follows: for every block that reaches the disk in the context of a certain file F, the delete of file F should trigger a secure overwrite (*i.e.*, *shred*) of the block. This behavior corresponds to the notion of *generation liveness* defined in Section 3. A *shred* involves multiple overwrites to the block with specific patterns so as to erase remnant magnetic effects of past layers (that could otherwise be recovered through techniques such as magnetic scanning tunneling microscopy [12]). Recent work suggests that two such overwrites are sufficient to ensure non-recoverability in modern disks [14].

Traditionally, secure deletion is implemented within the file system [3, 25, 26]; however, such implementations are unreliable given modern storage systems. First, for high security, overwrites need to be *off-track* writes (*i.e.*, writes straggling physical track boundaries), which external erase programs (*e.g.*, the file system) cannot perform [13]. Further, if the storage system buffers writes in NVRAM [32], multiple overwrites done by the file system may be collapsed into a single write to the physical disk, making the overwrites ineffective. Finally, in the presence of block migration [7, 32] within the storage system, an overwrite by the file system will only overwrite the current block location; stray copies of deleted data could remain. Thus, the storage system is the proper locale to implement secure deletion.

Note that FADED operates at the granularity of an entire volume; there is no control over which individual files are shredded. However, this limitation can be dealt with by storing "sensitive" files in a separate volume on which the secure delete functionality is enabled.

7.2 Basic operation

As discussed in Section 6.3, FADED monitors writes to inode and indirect blocks and tracks the inode generation to which each block belongs. It augments this information with the block liveness information it collects through the shadow bitmap technique. Note that since ext2 obeys the block exclusivity and delete suppression properties, block liveness detection is reliable. Thus, when a block death is detected, FADED can safely shred that block.

On the other hand, if FADED detects a generation death through the ownership change or generation change monitors (*i.e.*, the block is live according to the block liveness module), FADED cannot simply shred the block, because FADED does not know if the current contents of the block belong to the generation that was deleted, or to a new generation that was subsequently allocated the same block due to block reuse. If the current contents of the block are valid, a shredding of the block would be catastrophic.

We deal with such uncertainty through a conservative approach to generation-death inference. By being conservative, we convert an apparent correctness problem into a performance problem, *i.e.*, we may end up performing more overwrites than required. Fundamental to this approach is the notion of a *conservative overwrite*.

7.2.1 Conservative overwrites

A conservative overwrite of block B erases past layers of data on the block, but leaves the current contents of B intact. Thus, even if FADED does not know whether a subsequent valid write occurred after a predicted generation death, a conservative overwrite on block B will be safe; it can never shred valid data. To perform a conservative overwrite of block B, FADED reads the block B into non-volatile RAM, then performs a normal secure overwrite of the block with the specific pattern, and ultimately restores the original data back into block B.

The problem with a conservative overwrite is that if the block contents that are restored after the conservative overwrite are in fact the old data (which had to be shredded), the conservative overwrite was ineffective. In this case, FADED can be guaranteed to observe one of two things. First, if the block had been reused by the file system for another file, the new, valid data will be written eventually (*i.e.*, within the delayed write interval of the file system). When FADED receives this new write, it buffers the write, and before writing the new data to disk, FADED performs a shred of the concerned block once again; this time, FADED knows that it need not restore the old data, because it has the more recent contents of the block. To identify which writes to treat in this special manner, FADED tracks the list of blocks that were subjected to a conservative overwrite in a *suspicious blocks* list, and a write to a block in this list will be committed only after a secure overwrite of the block; after the second overwrite, the block is removed from the suspicious list. Note that the suspicious list needs to be stored persistently, perhaps in NVRAM, in order to survive crashes.

Second, if the block is not reused by the file system immediately, then FADED is guaranteed to observe a bitmap reset for the corresponding block, which will be flagged as a block death by the block liveness detector. Since block liveness tracking is reliable, FADED can now shred the block again, destroying the old data. Thus, in both cases of wrongful restore of old data, FADED is guaranteed to get another opportunity to make up for the error.

7.2.2 Cost of conservatism

Conservative overwrites come with a performance cost; every conservative overwrite results in the concerned block being treated as "suspicious", regardless of whether the data restored after the conservative overwrite was the old or new data, because FADED has no information to find it at that stage. Because of this uncertainty, even if the data restored were the new data (and hence need not be overwritten again), a subsequent write of the block in the context of the same file would lead to a redundant shredding of the block. Here we see one example of the performance cost FADED pays to circumvent the lack of perfect information.

7.3 Coverage of deletes

In the previous subsection, we showed that for all generation deaths detected, FADED ensures that the appropriate block version is overwritten, without compromising valid data. However, for FADED to achieve its goals, these detection techniques must be *sufficient* to identify *all* cases of deletes at the file system level that need to be shredded. In this section, we show that FADED can indeed detect all deletes, but requires two minor modifications to ext2.

7.3.1 Undetectable deletes

Because of the weak properties of ext2, certain deletes can be missed by FADED. We present the two specific situations where identification of deletes is impossible, and then propose minor changes to ext2 to fix those scenarios.

File truncates: The generation change monitor assumes that the version number of the inode is incremented when the inode is reused. However, the version number in ext2

Operation	In-memory	On-disk
Initial	$I_1 \rightarrow B^{Ind}$	$I_1 \rightarrow B^{Ind}$
I_1 delete	B free	
I_2 alloc	$I_2 \rightarrow B$	
B write to disk		$I_1 \rightarrow B^{Ind}$ (wrong type)

Table 6: **Misclassified indirect block.** *The table shows a scenario where a normal data block is misclassified as an indirect block. B^{Ind} indicates that B is treated as an indirect block. Reuse ordering for indirect blocks prevents this problem.*

is only incremented on a complete delete and reuse; partial truncates do not affect the version number. Thus if a block is freed due to a partial truncate and is reassigned to the same file, FADED misses the generation death. Although such a reuse after a partial truncate could be argued as a logical overwrite of the file (and thus, not a *delete*), we adopt the more complex (and conservative) interpretation of treating it as a delete.

To handle such deletes, we propose a small change to ext2; instead of incrementing the version number on a reallocation of the inode, we increment it on every truncate. Alternatively, we could introduce a separate field to the inode that tracks this version information. This is a non-intrusive change, but is effective at providing the disk with the requisite information. This technique could result in extra overwrites in the rare case of partial truncates, but correctness is guaranteed because the "spurious" overwrites would be conservative and would leave data intact.

Reuse of indirect blocks: A more subtle problem arises due to the presence of indirect pointer blocks. Indirect blocks share the data region of the file system with other user data blocks; thus the file system can reuse a normal user data block as an indirect block and vice versa. In the presence of such *dynamic typing*, the disk cannot reliably identify an indirect block [23].

The only way FADED can identify a block B as an indirect block is when it observes an inode I_1 that contains B in its indirect pointer field. FADED then records the fact that B is an indirect block. However, when it later observes a write to B, FADED cannot be certain that the contents indeed are those of the indirect block, because in the meanwhile I_1 could have been deleted, and B could have been reused as a user data block in a different inode I_2. This scenario is illustrated in Table 6.

Thus, FADED cannot trust the block pointers in a suspected indirect block; this uncertainty can lead to missed deletes in certain cases. To prevent this occurrence, a data block should never be misclassified as an indirect block. To ensure this, before the file system allocates, and immediately after the file system frees an indirect block B^{Ind}, the concerned data bitmap block $M_{B^{Ind}}$ should be flushed to disk, so that the disk will know that the block was freed. Note that this is a weak form of reuse ordering only for indirect blocks. As we show later, this change has very little impact on performance, since indirect blocks tend to be a very small fraction of the set of data blocks.

Operation	In-memory	On-disk
Initial	B free	B free
I_1 alloc	$I_1 \rightarrow B$	
B write to disk		B written
I_1 delete	B free	
I_2 alloc	$I_2 \rightarrow B$	
I_2 write to disk		$I_2 \rightarrow B$ (Missed delete of B)

Table 7: **Missed delete due to an orphan write.** *The table illustrates how a delete can be missed if an orphan block is not treated carefully. Block B, initially free, is allocated to I_1 in memory. Before I_1 is written to disk, B is written. I_1 is then deleted and B reallocated to I_2. When I_2 is written, FADED would associate B with I_2 and would miss the overwrite of B.*

Practicality of the changes: The two changes discussed above are minimal and non-intrusive; the changes together required modification of 12 lines of code in ext2. Moreover, they are required only because of the weak ordering guarantees of ext2. In file systems such as ext3 which exhibit reuse ordering, these changes are not required. Our study of ext2 is aimed as a limit study of the minimal set of file system properties required to reliably implement secure deletion at the disk.

7.3.2 Orphan allocations

Implicit block liveness tracking in FADED already addresses the orphan allocation issue discussed in § 5.3; when ext2 recovers after a crash, the fsck utility writes out a copy of all bitmap blocks; the block liveness monitor in FADED will thus detect death of those orphan allocations.

7.3.3 Orphan writes

Due to arbitrary ordering in ext2, FADED can observe a write to a newly allocated data block before it observes the corresponding owning inode. Such *orphan writes* need to be treated carefully because if the owning inode is deleted before being written to disk, FADED will never know that the block once belonged to that inode. If the block is reused in another inode, FADED would miss overwriting the concerned block which was written in the context of the old inode. Table 7 depicts such a scenario.

One way to address this problem is to *defer* orphan block writes until FADED observes an owning inode [23], a potentially memory-intensive solution. Instead, we use the suspicious block list used in conservative overwrites to also track orphan blocks. When FADED observes a write to an orphan block B, it marks B suspicious; when a subsequent write arrives to B, the old contents are shredded. Thus, if the inode owning the block is deleted before reaching disk, the next write of the block in the context of the new file will trigger the shred. If the block is not reused, the bitmap reset will indicate the delete.

This technique results in a redundant secure overwrite anytime an orphaned block is overwritten by the file sys-

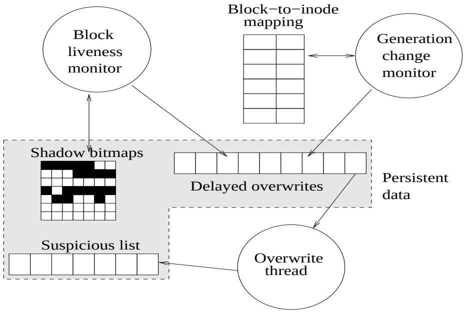


Figure 1: **Key components of FADED.**

tem in the context of the same file, again a cost we pay for conservatism. Note that this overhead is incurred only the first time an orphan block is overwritten.

7.3.4 Guaranteed detection of deletes

With these techniques, we can prove that for every block B that is deleted by the file system after it has reached disk, FADED always overwrites the deleted contents of B. The proof is presented in Appendix A.

7.4 Delayed overwrites

Multiple overwrites of the same block cause additional disk I/Os that can hurt performance if incurred on the critical path. For better performance, FADED delays overwrites until idle time in the workload [11] (or optionally, until up to n minutes of detection). Thus, whenever FADED decides to shred a block, it just queues it; a low priority thread services this queue if FADED had not observed useful foreground traffic for more than a certain duration. Delayed overwrites help FADED to present writes to the disk in a better, sequential ordering, besides reducing the impact on foreground performance. Delaying also reduces the number of overwrites if the same block is deleted multiple times. The notion of conservative overwrites is crucial to delaying overwrites arbitrarily, even after the block that had to be overwritten is written in the context of a new file. Note that if immediate shredding is required, the user needs to perform a `sync`.

A summary of the key data structures and components of FADED is presented in Figure 1.

7.5 FADED for other file systems

We have also implemented FADED underneath other file systems, and in each case, validated our implementation with the same testing methodology as will be described in Section 7.7. However, due to space constraints, we only point to the key differences we observed relative to ext2.

7.5.1 FADED for VFAT

Like ext2, VFAT also does not conform to reuse ordering, so FADED needs to track generation information for each block in order to detect deletes. One key difference in VFAT compared to ext2 is that there are no pre-allocated, uniquely addressable "inodes", and consequently, no "version" information as well. Dynamically allocated directory blocks contain a pointer to the start block of a file; the FAT chains the start block to the other blocks of the file. Thus, detecting deletes reliably underneath unmodified VFAT is impossible. We therefore introduced an additional field to a VFAT directory entry that tracks a globally unique *generation number*. The generation number gets incremented on every create and delete in the file system, and a newly created file is assigned the current value of generation number. With this small change (29 lines of code) to VFAT, the generation change monitor accurately detects all deletes of interest.

7.5.2 FADED for ext3

Since ext3 exhibits reuse ordering, tracking generation liveness in ext3 is the same as tracking block liveness. However, since ext3 does not obey the block exclusivity property, tracking block liveness accurately is impossible except in the data journaling mode which has the useful property of data-metadata coupling. For the ordered and writeback modes, we had to make a small change: when a metadata transaction is logged, we also made ext3 log a list of data blocks that were allocated in the transaction. This change (95 lines of code), coupled with the reuse ordering property, enables accurate tracking of deletes.

7.6 Explicit secure delete

We have also built secure deletion under the explicit notification framework. We modified the ext2 and ext3 file systems to notify the disk of every logical delete (as described in §5). The file system modifications accounted for 14 and 260 lines of code respectively. Upon receiving the notification, the disk decides to shred the block. However, similar to FADED, the disk delays overwrites until idle time to minimize impact on foreground performance.

7.7 Evaluation

In this section, we evaluate our implicit and explicit implementations of secure delete. The enhanced disk is implemented as a pseudo-device driver in the Linux 2.4 kernel; the driver observes the same information as a hardware prototype, but suffers contention for CPU and memory from the host. We use a 2.4 GHz Pentium-4 with 1 GB RAM and a 10K RPM IBM 9LZX disk. Due to space constraints, we provide results only for the ext2 version.

7.7.1 Correctness and accuracy

To test whether our FADED implementation detected all deletes of interest, we instrument the file system to log every delete, and correlate it with the log of writes and overwrites by FADED, to capture cases of unnecessary or missed overwrites. We tested our system on various workloads with this technique, including a few busy hours from the HP file system traces [19]. Table 8 presents the results of this study on the trace hour $09\ 00$ of $11/30/00$.

In this experiment, we ran FADED under four versions of Linux ext2. In the first, marked "No changes", a default

Config	Delete	Overwrite	Excess	Miss
No changes	76948	68700	11393	854
Indirect	76948	68289	10414	28
Version	76948	69560	11820	0
Both	76948	67826	9610	0

Table 8: **Correctness and accuracy.** *The table shows the number of overwrites performed by the FADED under various configurations of ext2. The columns (in order) indicate the number of blocks deleted within the file system, the total number of logical overwrites performed by FADED, the number of unnecessary overwrites, and the number of overwrites missed by FADED. Note that deletes that occurred before the corresponding data write do not require an overwrite.*

Config	Reads	Writes	Run-time(s)
No changes	394971	234664	195.0
Version	394931	234648	195.5
Both	394899	235031	200.0

Table 9: **Impact of FS changes on performance.** *The performance of the various file system configurations under a busy hour of the HP Trace is shown. For each configuration, we show the number of blocks read and written, and the trace run-time.*

ext2 file system was used. In "Indirect", we used ext2 modified to obey reuse ordering for indirect blocks. In "Version", we used ext2 modified to increment the inode version number on every truncate, and the "Both" configuration represents both changes (the correct file system implementation required for FADED). The third column gives a measure of the extra work FADED does in order to cope with inaccurate information. The last column indicates the number of missed overwrites; in a correct system, the fourth column should be zero.

We can see that the cost of inaccuracy is quite reasonable; FADED performs roughly 14% more overwrites than the minimal amount. Also note that without the version number modification to ext2, FADED indeed misses a few deletes. The reason no missed overwrites are reported for the "Version" configuration is the rarity of the case involving a misclassified indirect block.

7.7.2 Performance impact of FS changes

We next evaluate the performance impact of the two changes we made to ext2, by running the same HP trace on different versions of ext2. Table 9 shows the results. As can be seen, even with both changes, the performance reduction is only about 2% and the number of blocks written is marginally higher due to synchronous bitmap writes for indirect block reuse ordering. We thus conclude that the changes are quite practical.

7.7.3 Performance of secure delete

We now explore the foreground performance of implicit and explicit secure delete, and the cost of overwrites.

Foreground performance impact: Tracking block and generation liveness requires FADED to perform extra processing. This cost of reverse engineering directly impacts application performance because it is incurred on the critical path of every disk operation. We quantify the impact of this extra processing required at FADED on foreground performance. Since our software prototype competes for CPU and memory resources with the host, these are worst case estimates of the overheads.

System	Run-time (s)		
	Implicit		Explicit
	PostMark	HP Trace	HP Trace
Default	166.8	200.0	195.0
SecureDelete$_2$	177.7	209.6	195.5
SecureDelete$_4$	178.4	209.0	196.8
SecureDelete$_6$	179.0	209.3	196.4

Table 10: **Foreground impact: Postmark and HP trace.** *The run-times for Postmark and the HP trace are shown for FADED, with 2, 4 and 6 overwrite passes. For comparison, the run-time of explicit secure delete on the HP Trace is also shown. Postmark was configured with 40K files and 40K transactions.*

We run the Postmark file system benchmark [15] and the HP trace on a file system running on top of FADED. Postmark is a metadata intensive small-file benchmark, and thus heavily exercises the inferencing mechanisms of FADED. To arrive at a pessimistic estimate, we perform a `sync` at the end of each phase of Postmark, causing all disk writes to complete and account that time in our results. Note that we do not wait for completion of delayed overwrites. Thus, the numbers indicate the performance perceived by the foreground task.

Table 10 compares the performance of FADED both with a default disk and with explicit secure delete. From the table, we can see that even for 4 or 6 overwrite passes, foreground performance is not affected much. Extra CPU processing within FADED causes only about 4 to 7% lower performance compared to the modified file system running on a normal disk. The explicit implementation performs better because it does not incur the overhead of inference. Further, it does not require the file system modifications reported in Table 9 (this corresponds to the "No changes" row in Table 9). Note that we do not model the cost of sending a `free` command across the SCSI bus; thus the overheads in the explicit case are optimistic.

Idle time required: We now quantify the cost of performing overwrites for shredding. With microbenchmarks, we verified that the overwrites obtained near sequential bandwidth due to their delayed, ordered issue. We also found that when block reuse occurs within the file system (resulting in multiple deletes to the same block), delaying overwrites significantly reduces overwrite traffic. We omit these results due to space constraints.

We next explore the time required for overwrites. First, we use the same Postmark configuration as above, but measure the time for the benchmark to complete including delayed overwrites. Since Postmark deletes all files at the end of its run, we face a worst case scenario where the entire working set of the benchmark has to be overwrit-

System	Run-time with overwrites (s)		
	Implicit		Explicit
	PostMark	HP Trace	HP Trace
Default	166.8	200.0	195.0
SecureDelete$_2$	466.6	302.8	280.0
SecureDelete$_4$	626.4	345.6	316.2
SecureDelete$_6$	789.3	394.3	346.1

Table 11: **Idle time requirement.** *The table shows the total run-time of two benchmarks, Postmark and the HP trace. The time reported includes completion of all delayed overwrites.*

ten, accounting for the large overwrite times reported in Table 11. In the HP-trace, the overwrite times are more reasonable. Since most blocks deleted in the HP trace are then reused in subsequent writes, most of the overwrites performed here are conservative. This accounts for the steep increase from 0 to 2 overwrite passes, in the implicit case. The explicit implementation incurs 8-13% lower overwrite times compared to FADED because it has perfect information on deletes, and thus avoids extra overwrites incurred due to conservatism.

8 Implicit Detection Under NTFS

In this section, we present our experience building support for implicit liveness detection underneath the Windows NTFS file system. The main challenge we faced underneath NTFS was the absence of source code for the file system. While the basic on-disk format of NTFS is known [27], details of its update semantics and journaling behavior are not publicly available. As a result, our implementation currently tracks only block liveness which requires only knowledge of the on-disk layout; generation liveness tracking could be implemented if the details of NTFS journaling mechanism were known.

The fundamental piece of metadata in NTFS is the Master File Table (MFT); each record in the MFT contains information about a unique file. Every piece of metadata in NTFS is treated as a regular file; file 0 is the MFT itself, file 2 is the recovery log, and so on. The allocation status of all blocks in the volume is maintained in a file called the cluster bitmap, which is similar to the block bitmap tracked by ext2. On block allocations and deletions, NTFS regularly writes out modified bitmap blocks.

Our prototype implementation runs as a device driver in Linux, similar to the setup described earlier for other file systems. The virtual disk on which we interpose is exported as a logical disk to a virtual machine instance of Windows XP running over VMware Workstation [30]. To track block liveness, our implementation uses the same shadow bitmap technique mentioned in Section 6.2. By detailed empirical observation under long-running workloads, we found that NTFS did not exhibit any violation of the block exclusivity and delete suppression properties mentioned in Section 4.2; however, due to the absence of source code, we cannot assert that NTFS *always* conforms to these properties. This limitation points to the general difficulty of using implicit techniques underneath closed-source file systems; one can never be certain that the file system conforms to certain properties unless those are guaranteed by the file system vendor. In the absence of such guarantees, the utility of implicit techniques is limited to optimizations that can afford to be occasionally "wrong" in their implicit inference.

Our experience with NTFS also points to the utility of characterizing the precise set of file system properties required for various forms of liveness inference. This set of properties now constitutes a minimal "interface" for communication between file system and storage vendors. For example, if NTFS confirmed its conformance to the block exclusivity and delete suppression properties, the storage system could safely implement aggressive optimizations that rely on its implicit inference.

9 Discussion

In this section, we reflect on the lessons learned from our case study to refine our comparison on the strengths and weaknesses of the explicit and implicit approaches.

The ideal scenario for the implicit approach is where changes are required only in the storage system and not in the file system or the interface. However, in practice, accurate liveness detection requires certain file system properties, which means the file system needs to be modified if it does not conform to those requisite properties. In the face of such changes to both the storage system and the file system, it might appear that the implicit approach is not much more pragmatic than the explicit approach of changing the interface also. There are two main reasons why we believe the implicit approach is still useful.

First, file system changes are not required if the file system already conforms to the requisite properties. For example, many file systems (e.g. ext2, VFAT, ext3-data journaling, and perhaps NTFS) are already amenable to block liveness detection without any change to the file system. The ext3 file system in data journaling mode already conforms to the properties required for generation liveness detection. Clearly, in such cases, the implicit approach enables non-intrusive deployment of functionality.

Second, we believe that modifying the file system to conform to a set of well-defined properties is more general than modifying the file system (and the interface) to convey a specific piece of information. Although we have discussed the file system properties from the viewpoint of implicit liveness detection, some of the properties enable richer information to be inferred; for example, the association between a block and its owning inode (required for certain applications such as file-aware layout [23]) can be tracked accurately if the file system obeys the reuse ordering or the consistent metadata properties. Our ultimate goal is to arrive at a set of properties that enable a wide variety of information to be tracked implicitly, thus outlining how file systems may need to be designed to enable such transparent extension within the storage system. In contrast, the approach of changing the interface requires introducing a new interface every time a different piece of information is required.

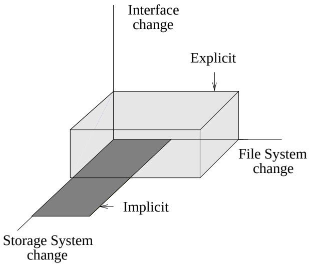
10 Related Work

The need for liveness information in storage systems has been recognized in previous work. In most existing proposals, an interface to communicate liveness is a part of a more radical set of changes to the existing storage interface. For example, logical disks have a list-based interface to storage which includes a command to "delete" a block from a list [4]. More recent work suggests an object-like interface to storage [17], which moves the responsibilities of low-level storage management such as liveness tracking from the file system into the drives themselves. In contrast to such wide-scale changes, our "explicit notification" approach for imparting liveness is much less intrusive on the large body of file systems that utilize the existing block-based interface to storage.

There has also been work on implementing "smarts" within a storage system without interface change, similar to our implicit approach. Some of these systems utilize a limited form of liveness inference. For example, AutoRAID requires information on free space to decide the amount of data that can be stored in RAID-1 [32]; AutoRAID infers that blocks that have not been written ever, are dead. This inference is a weak form of liveness because once a block is written, subsequent deletes cannot be detected. Other systems such as the programmable disk [31] make similar inferences. The existence of these proposals indicates that liveness information is important in storage systems, and yet systematic techniques for acquiring such information have been missing.

Most related to the implicit techniques in this work is our previous work on semantically-smart disks [24]. In that work, we presented techniques by which a block-based storage system can infer file system level information and implemented a set of case studies such as track-aligned extents, journaling, and secure delete. However, all correctness-sensitive case studies implemented therein required the file system to be synchronously mounted; under synchronous file systems, implicit information tracking is trivial. Our more recent work on D-GRAID [23] considered asynchronous file systems, but the layout mechanisms of D-GRAID did not depend on accuracy for correctness; it was acceptable in D-GRAID to get predictions wrong. Also, fast recovery in D-GRAID utilized *block liveness* (a much easier property to track than generation liveness) under specific assumptions on file system behavior. In this work, we go beyond our previous work by generalizing our techniques for inference underneath a wide range of realistic file system behaviors, and demonstrating that storage-level functionality where correctness is paramount, can utilize this information reliably.

11 Conclusion



As system layers evolve over time, interfaces between layers become obsolete or sub-optimal, necessitating their evolution. We have presented two approaches for interface evolution: explicit and implicit, in the context of embedding liveness information into storage. A qualitative summary of the complexity of the two approaches along various axes is presented in the figure. We have shown that the explicit approach, while appearing straightforward, entails a fair amount of file system change in practice, besides requiring some minimal support from the storage system. Despite these factors, the explicit approach results in simpler systems than the implicit case. The main strength of the implicit approach is that it permits demonstration of functionality without changes to the interface, thus enabling seamless deployment while catalyzing rapid interface evolution.

Acknowledgments

We thank Nitin Agrawal, John Bent, Timothy Denehy, Todd Jones, James Nugent, Florentina Popovici, and Vinod Yegneswaran for their helpful comments. We also thank Mendel Rosenblum for his excellent shepherding, the anonymous reviewers for their thoughtful feedback, and Gordon Hughes for his useful comments on secure delete. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM and EMC.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th VLDB*, 2002.

[2] L. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *ISCA '04*, 2004.

[3] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *USENIX Security*, August 2001.

[4] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *SOSP '93*, 1993.

[5] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX*, Monterey, CA, June 2002.

[6] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *FREENIX*, June 2002.

[7] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[8] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *USENIX*, Jan. 1992.

[9] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. TR SCS CMU-CS-01-166, Dec. 2001.

[10] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 18(2), May 2000.

[11] R. A. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.

[12] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security*, July 1996.

[13] G. Hughes. Personal communication, 2004.

[14] G. Hughes and T. Coughlin. Secure Erase of Disk Drive Data. IDEMA Insight Magazine, 2002.

[15] J. Katcher. PostMark: A New File System Benchmark. NetApp TR-3022, October 1997.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *TOCS*, 2(3), Aug. 1984.

[17] M. Mesnier, G. R. Ganger, and E. Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41(8), August 2003.

[18] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based Intrusion Detection: Watching Storage Activity For Suspicious Behavior. In *USENIX Security*, 2003.

[19] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, 2002.

[20] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, 2000.

[21] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, HP Laboratories, 1991.

[22] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *FAST '02*, January 2002.

[23] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*, Mar. 2004.

[24] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*, 2003.

[25] SourceForge. SRM: Secure File Deletion for POSIX Systems. http://srm.sourceforge.net, 2003.

[26] SourceForge. Wipe: Secure File Deletion. http://wipe.sourceforge.net, 2003.

[27] SourceForge. The Linux NTFS Project. http://linux-ntfs.sf.net/, 2004.

[28] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *OSDI 2000*, 2000.

[29] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX*, June 2002.

[30] VMWare. VMWare Workstation 4.5. http://www.vmware.com/products/, 2004.

[31] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *OSDI '99*, 1999.

[32] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[33] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading Capacity for Performance in a Disk Array. In *OSDI '00*, 2000.

A Guaranteed detection of deletes

We now prove that the techniques in FADED for ext2 guarantee shredding of all deletes of blocks whose contents reached disk.

When a delete of an inode I_1 occurs within ext2, a set of blocks are freed from a file; this results in an increment of the version number of I_1, and the reset of relevant bits in the data bitmap block pertaining to the freed blocks. Let us consider one such block B that is freed. Let us assume that B had already been written to disk in the context of I_1. If B had not been written to disk, the disk does not need to perform any overwrite, so we do not consider that case. Let the bitmap block containing the status of B be M_B, and let B_I be the block containing the inode I_1. Now, there are two possibilities: either B is reused by the file system *before* M_B is written to disk, or B is not reused until the write of M_B.

Case 1: Block B not reused

If B is not reused immediately to a different file, the bitmap block M_B, which is dirtied, will be eventually written to disk, and the disk will immediately know of the delete through the block liveness module, and thus overwrite B.

Case 2: Block B is reused

Let us now consider the case where B is reused in inode I_2. There are three possibilities in this case: *at the point of receiving the write of* B, the disk either thinks B belongs to I_1, or it thinks B is free, or that B belongs to some other inode I_x.

Case 2a: Disk thinks $I_1 \rightarrow B$

If the disk knew that $I_1 \rightarrow B$, the disk would have tracked the previous version number of I_1. Thus, when it eventually observes a write of B_I, (which it will, since B_I is dirtied because of the version number increment), the disk will note that the version number of I_1 has increased, and thus would overwrite all blocks that it thought belonged to I_1, which in this case includes B. Thus B would be overwritten, perhaps restoring a newer value. As discussed in Section 7.2, even if this was a conservative overwrite, the old contents are guaranteed to be shredded.

Case 2b: Disk thinks B is free

If the disk thinks B is free, it would treat B as an orphan block when it is written, and mark it suspicious. Consequently, when B is written again in the context of the new inode I_2, the old contents of B will be shredded.

Case 2c: Disk thinks $I_x \rightarrow B$

To believe that $I_x \rightarrow B$, the disk should have observed I_x pointing to B at some point before the current write to B. [1] The disk could have observed $I_x \rightarrow B$ either before or after B was allocated to I_1 by the file system.

Case 2c-i: $I_x \rightarrow B$ before $I_1 \rightarrow B$

If the disk observed $I_x \rightarrow B$ *before* it was allocated to I_1, and still thinks $I_x \rightarrow B$ when B is written in the context of I_1, it means the disk never saw $I_1 \rightarrow B$. However, in this case, block B was clearly deleted from I_x at some time in the past in order to be allocated to I_1. This would have led to the version number of I_x incrementing, and thus when the disk observes I_x written again, it would perform an overwrite of B since it thinks B used to belong to I_x.

Case 2c-ii: $I_x \rightarrow B$ after $I_1 \rightarrow B$

If this occurs, it means I_x was written to disk owning B after B was deleted from I_1 but before B is written. In this case, B will only be written in the context of I_x which is still live, so it does not have to be overwritten. As discussed in Section 4.2, this holds because of the block exclusivity property of ext2.

Note that the case of a block being deleted from a file and then quickly reallocated to the same file is just a special case of *Case 2c* , with $I_1 = I_x$.

Thus, in all cases where a block was written to disk in the context of a certain file, the delete of the block from the file will lead to a shred of the deleted contents. □

[1] If indirect block detection was uncertain, the disk can wrongly think $I_x \rightarrow B$ because of a corrupt "pointer" in a false indirect block; our file system change for reuse ordering in indirect blocks prevents this case.