Towards Regulatory Compliant Storage Systems

Graduate Board Orals Proposal

Zachary N. J. Peterson

Department of Computer Science
The Johns Hopkins University
zachary@cs.jhu.edu

1 Introduction

Legislators and the courts have begun to recognize the importance of how electronically stored data should be maintained and secured, and how electronic data should be differentiated from their paper analogs. Examples of some of the sweeping pieces of electronic record management legislation include the Health Insurance Portability and Accountability Act (HIPAA) of 1996, the Gramm-Leach-Bliley Act (GLBA) of 1999, and the more recent Federal Information Security Management Act (FISMA) and Sarbanes-Oxley Act (SOX) of 2002. Altogether, there exist over 4,000 acts and regulations that govern digital storage, all with a varying range of requirements for maintaining electronic records.

Storage systems vendors have quickly identified the large market opportunity and have modified existing systems and marketed them as compliance products. Sarbanes-Oxley compliance alone represents a market of over $5 billion [22]. Mostly, vendors add policy enhancements to existing storage platforms that aid in the maintenance and retention of data, such as forbidding data deletion. Current product offerings include EMC Centera Compliance Edition, HP Reference Information Storage Systems, and IBM Tivoli Security Compliance Manager.

Many of these products fail to meet the new demands legislation places on storage systems. Systems must now provide confidentiality through encrypted storage and data transmission. Some legislation requires an auditable trail of changes made to electronic records that are accessible in real-time. This implies versioning files and providing a means of quickly retrieving versions from any point in time. Other legislation sets limits on the amount of time an organization may be liable for maintaining their electronic data, but for those data that go out of scope, permanently deleting data from magnetic media can be challenging. Because electronic data is dynamic, and therefore easily malleable on disk, new methods for authentication and non-repudiation need to be developed to ensure a binding of an individual to an auditable trail of data changes. Further, these systems must be robust against both external and internal attacks. A data loss or compromise due to negligence may result in an organization falling out of compliance and susceptible to litigation.

In this proposal we introduce some completed technical contributions to the field of regulatory compliant storage, and propose a set of goals pursuant to completing a Ph.D. We begin with a treatment of the ext3cow file system; an open-source versioning file system designed to be a platform for developing regulatory compliant storage technologies.

2 Research Foundations - The Ext3cow File System

As a first step towards creating a regulatory compliant storage system, we have developed *ext3cow* [42], an open-source, file versioning and snapshot file system. Ext3cow was originally designed to address the versioning and auditability needs of regulated storage, but has grown into a platform for developing technical solutions to a wide array of regulatory storage problems. Our model is designed to satisfy many of the federal requirements set forth by law, including: confidentiality, non-repudiation, authentication, as well as a tamper proof audit trail of data changes, providing a "chain of trust."

Ext3cow is built from the popular ext3 file system. Ext3 [11] is based on the Minix file system [58] and influenced by the Fast File System (FFS) [36]. Ext3 has become robust and reliable, providing reasonable performance and scalability for many users and workloads [10]. Ext3 is default file system on most Linux distributions. Ext3cow extends the ext3 design by retrofitting the in-memory and on-disk metadata structures to support lightweight, logical file systems snapshots and individual file versioning. All files and snapshots are available at all times, and ext3cow offers a fine-grained user-interface to access individual file and directory versions from snapshots.

Ext3cow differs from other efforts at versioning file systems in its combination of features. Ext3cow both (1) encapsulates all versioning function within the on-disk file systems and (2) provides a fine-grained, interactive, and continuous-time interface to file versions and snapshots. We accomplish this through a *time-shifting* interface. Time-shifting allows users and applications to interact directly with the disk file system, *i.e.* the interface is transparent to kernel components, in particular, the virtual file system. Many file systems that provide fine-grained access to versions do so by modifying kernel interfaces [14, 40, 51]. This incurs copying overheads, pollutes the buffer pool with old data, and complicates installation and management. Other disk file systems provide coarse-grained access to versions through the creation of namespace tunnels [25] or via mounting separate logical volumes [56, 57]. Some disk file systems provide no interface to versions, restricting versioning to internal use only [49, 53].

In time-shifting, ext3cow introduces an interface to versioning that presents a continuous view of time. Users or applications specify a file name and any point in time, which ext3cow scopes to the appropriate snapshot or file version. The time-shifting interface allows user-space tools to create snapshots and access versions. Applications may access these tools to coordinate snapshots with application state. User-space tools are suitable for automation, using software as simple as `cron`. Ext3cow's time-shifting and controlled versioning facilitates the consistent transfer of data from ext3cow to other storage systems.

Many of the virtues of ext3cow lie in encapsulating snapshot and versioning entirely within the on-disk file system. Ext3cow does not change any kernel interfaces and does not modify the common file object model provided by the virtual file system (VFS) [31]. This makes ext3cow easy to install in existing systems; it may be loaded as a module to a running kernel and co-exist with all other Linux file systems. Only an on-disk file system, such as ext3cow, can control data placement, metadata organization, and I/O. Specifically, ext3cow retains tight control on the versioning of buffers and pages. Ext3cow does not degrade cache performance by insuring the Linux page cache sees a single copy of file data; old versions of data exist only on disk. Copies are created on-demand when performing I/O to the disk. This is not possible in VFS implementations. Further, ext3cow's inode versioning policy maintains *stable inodes*, preserving a files inode number over the lifetime of a file. Because of stable inodes, ext3cow implicitly supports the Network File System (NFS [37, 50]). NFS file handles are essential to its stateless operation and require the inode numbers to remain the same over the lifetime of a file handle. Again, this is not possible in VFS implementations.

Lastly, some versioning systems require specialized, and often expensive hardware, making these sys-

tems unattractive for the consumer. Regulatory compliance places a tremendous financial burden on organizations. AMR research estimates the total spending on Sarbanes-Oxley compliance alone in 2004 to exceed $5 billion [22]. Experience with HIPAA [30] indicates that the costs of compliance are relatively greater for smaller organizations. This research is a key component in reducing the cost of compliance for small organizations. By providing an open-source system that satisfies the requirements of many electronic record management regulations, ext3cow will be particularly helpful to non-profits subject to government reporting requirements, small businesses subject to Sarbanes-Oxley, and small health care providers subject to HIPAA.

We have released ext3cow under the GNU Public License via *http://www.ext3cow.com*. As of this writing, ext3cow has had over a thousand visitors and hundreds of downloads from over one hundred different countries. We run a development mailing list to which a number of enthusiasts have subscribed. The authors have been running ext3cow to store data on their laptops and personal workstations since June 2003. We have not experienced a system crash or data loss incident in that period. Ext3cow has appeal beyond the regulatory environment for which it is designed; it has been adopted as the storage platform for several research projects.

2.1 Related Work

Storage and file systems use data versioning to enhance reliability, availability, and operational semantics. Versioning techniques include volume and file system snapshot as well as per-file versioning. A snapshot is a read-only, immutable, and logical image of a collection of data as it appeared at a single point in time. Point-in-time snapshots of a file system are useful for consistent image for backup [13, 19, 23, 26] and for archiving and data mining [45]. File versioning, creating new logical versions on every disk write or on every open/close session, is used for tamper-resistant storage [56,57] and file-oriented recovery from deletion [14, 40,51]. Both techniques speed recovery and limit exposure to data losses during file system failure [25,53]. A range of snapshot implementations exist, both at the logical file system level [19, 26, 27, 45, 51] and the disk storage level [17,24,27,57].

File system versioning and snapshot have been used to recover from failure. FFS [33, 35, 36] takes snapshots to create a quiescent file system on which to perform on-line file system integrity checking. FFS does not provide an interface to access file snapshots on-line. WAFL [25] also uses snapshots for recovery. It provides users a `.snapshot` directory for every directory in the file system containing discrete views of the past.

File system snapshots implemented with copy-on-write are an implicit feature of log-structured file systems. LFS [49, 53] and Spiralog [20, 28] do not overwrite file data as they are written, but instead write changes as they occur to a circular log. Checkpoints, which serve as snapshots in log-structured file systems, are used to roll-back a file system to a known consistent point after a system failure. LFS and Spiralog do not provide an interface to access versions.

The Andrew file system [26,38], the Episode file system [13], Plan-9 [43,44], and SnapMirror [41] use snapshot with copy-on-write techniques as a method to perform quick, low-bandwidth backups in an on-line fashion. Venti [45] uses hashing and copy-on-write to archive blocks efficiently. A survey and evaluation of snapshot and backup techniques was performed by Chervenak *et al* [12] and Azagury *et al* [1].

Cedar [19,23,52] is the first example of a file system that maintains versions of a file over time. Versions are shared among file system users. Similar approaches were used by VMS [15,32] and TOPS [39].

The Elephant file system [51] is the first file system to include a variety of user-specified retention policies similar to user-space version control tools. Wayback [14] uses a similar versioning paradigm.

	ext3cow	**CVFS**	**Elephant**	**Wayback**	**WAFL**	**LFS**
Disk file system	•				•	•
Preserves kernel interfaces	•			•	N/A[1]	•
Files system snapshot	•	•			•	•
File versioning	•	•	•	•		
Time-oriented interface	•		•			
Preserves FS namespace	•	•	•			•
Stable inodes for NFS	•	•			•	•
Open-source license	•			•		•

Table 1: Feature comparison of versioning file systems.

```
[user@machine] echo "This is the original foo.txt" >
foo.txt
[user@machine] snapshot
Snapshot on .  1057845484
[user@machine] echo "This is the new foo.txt." >
foo.txt
[user@machine] cat foo@1057845484
This is the original foo.txt.
[user@machine] cat foo
This is the new foo.txt.
```

Figure 1: Creating snapshots and accessing data in the past in ext3cow.

In the Comprehensive Versioning File System (CVFS) [56], all writes to the server, in a client/server storage system, are versioned, which provides an audit trail for security breaches.

To place our contributions in context with respect to recent versioning file system research, Table 1 compares the features of ext3cow to CVFS [56], Elephant [51], Wayback [14], WAFL [25], and log-structured file systems (LFS) [49, 53]. We restrict this treatment to file systems, omitting versioning archives [2, 45], because we are concerned with interactive versioning in the regulatory environment. We also omit VersionFS [40] because it compares similarly to Wayback. This table punctuates our contribution. Ext3cow provides the benefits of fine-grained versioning with interactive, real time access to versions, without manipulating kernel interfaces.

2.2 Time-shifting

Our goals in creating an interface to data versioning include offering rich semantics, making it congruent with operating system kernel interfaces, and providing access to all versions from within the file system. Semantically rich means that the way in which data are accessed provides insight into the age of the data. In the time-shifting principle, date and time information are embedded into the access path. The interface allows a user to fetch any file or directory from any point in time and to navigate the file system in the past.

We describe the operation of the time shifting interface through the example of Figure 1. A call to the `snapshot` utility causes a snapshot of the file system to be taken and returns the *snapshot epoch* `1057845484`. For epoch numbers, we use the number of seconds since the Epoch (00:00:00 UTC, January

[1] WAFL is implemented as a file-system appliance within a custom operating system.

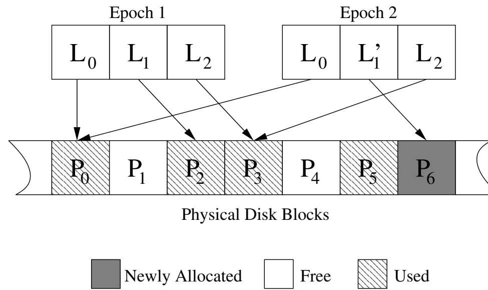


Figure 2: An example of copy-on-write. The version from epoch 2 updates logical block L_1 into L_1'. Ext3cow allocates a new physical (disk) block P_6 to record the difference. All other blocks are shared.

1, 1970), which may be acquired through `gettimeofday`. Subsequent writes to the file cause the current version to be updated, but the version of the file at the snapshot is unchanged. To access the snapshot version, a user or application appends the `@` symbol to the name and specifies a time. `Snapshot` is a user space program and library call that invokes a file-system specific `ioctl`, instructing ext3cow to create a snapshot. Using `ioctl` allows `snapshot` to bypass the virtual file system and communicate with ext3cow directly, which is consistent with our ethic of making no changes to the kernel.

We designed the time-shifting interface for applications and enhance its interactive usability through shell extensions. The number of seconds since the Epoch conforms to `gettimeofday` and is the natural way for applications to query, store, and encode time. However, humans prefer richer time formats, such as `[[[[[cc]yy]mm]dd]hh]mm[.ss]` in the `date` utility. To enhance usability for humans, we are developing shell extensions in a *time-traveling* csh (ttcsh), which will support a variety of date, time and naming formats to help users browse versions.

The time-shifting interface meets regulatory requirements. Users and applications specify a day, hour, and second at which they want a file. The interface does not require the specified time to be exactly on a snapshot. Rather, the interface treats time continuously. Requesting a file at a time returns the file contents at the preceding snapshot. The interface uses the `@` symbol, a legal symbol for file names, so that the VFS accepts the name and passes it through to ext3cow unmodified. The interface adds no new names to the namespace.

2.3 Versioning with Copy-on-Write

Ext3cow uses a *disk-oriented* copy-on-write scheme that supports file versioning without polluting Linux's page cache. Copies of data blocks exist only on disk and not in memory. This differs from other forms of copy-on-write used in operating systems that create two in-memory copies, such as process forking (`vfork` [34]) and the virtual memory management of shared pages. Ext3cow has the same memory footprint for data blocks as ext3, and, thus, does not incur overheads for copying pages or by using more memory, which reduces system cache performance.

Ext3cow employs the copy-on-write of file system blocks to implement multiple versions of data compactly. Scoping rules allow a single version of a file to span many epochs. Therefore, ext3cow needs to create a new physical version of a file only when data changes. Frequently, physical versions have much data in common. Copy-on-write allows versions to share a single copy of file system blocks for common data and have their own copy of blocks of data that have changed (Figure 2).

When the most recent version of a file precedes the system epoch in time, any change to that file creates

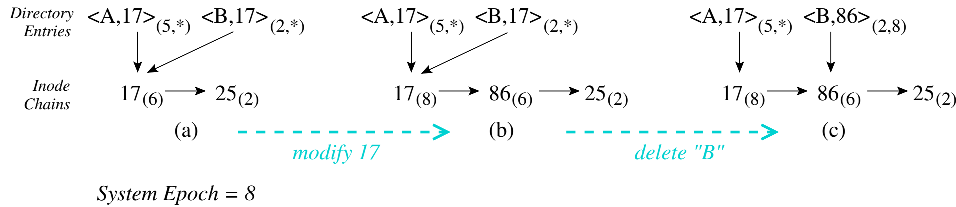


Figure 3: An example of names scoping to inodes over time.

a new physical version. The first step is to duplicate the inode, as discussed in Section 2.4. The duplicated inodes (new and old) initially share all data blocks in common. This includes sharing all indirect blocks, also doubly and triply indirect blocks. The first time that a logical block in the new file is updated, ext3cow allocates a new physical disk block to hold the data, preserving a copy of the old block for the old version. Subsequent updates to the same data in the same epoch are written in place; copy-on-write occurs at most once per epoch. Updates to data in indirect blocks (resp. doubly and triply indirect blocks), change not only data blocks, but also indirect blocks. Ext3cow allocates a new disk block as a copy-on-write version of an indirect block.

2.4 Version Scoping

Ext3cow maps point-in-time requests to snapshots and object versions through scoping metadata in directory entries and names. The logically continuous (to the second) time-shifting interface does not match exactly the realities of versioning. Several system properties govern ext3cow's versioning model. First, a version of file metadata or data covers a period of time; generally many different snapshot epochs. Also, ext3cow retains data at the time of a snapshot and does not track intermediate changes. When updating data or metadata, ext3cow marks versions with the current system epoch, not the current time. Finally, ext3cow maps point-in-time requests to the version preceding the exact time of the request. All told, this means that when accessing data in the past, all modifications that occur during an epoch are indivisible and occur at the start of an epoch.

A notable boundary case arises in the snapshot number returned by the `snapshot` utility. Intuitively, the snapshot number provides access to the file system at the time at which the snapshot was taken. `Snapshot` returns the current time and sets the system epoch counter to this value plus one. The return value, say j, provides a handle to all changes included in the previous epoch. The system sets the counter for the current epoch to $j+1$. The next snapshot taken at k covers the period $[j+1,k]$. Access to any time in this interval, including k, retrieves data marked with epoch $j+1$.

Scoping backward in time provides a natural interface for file versioning and recovery. For example, a user accidentally deletes a file at some time $t \geq k$, but remembers the file exists at some time $s \in [j+1,k]$. To restore the file, the user specifies s in the time-shifting interface, `file@s`. The enumeration of versions aids this process; users see all points-in-time at which the file changed using the `ls file@` command and can identify the desired file version.

2.4.1 Scoping Inodes

Ext3cow implements versioning by chaining together versions of inodes, a representation of a file's metadata. Each inode in the chain represents a specific version in time. Inode chains provide a continuous-time view of all versions of a file. The chain links inodes backward in time. To find an inode for a particular

epoch, ext3cow traverses the inode chain until it locates an inode with an epoch less than or equal to the requested point-in-time. At the head of the chain sits the most recent version of the inode. This design minimizes access latency to the current version – the most common operation. Figure 3(a) shows inode 17 last written during epoch 6. Subsequent to that write, a snapshot has been taken, indicated by the system epoch counter value of 8. A modification to inode 17 (Figure 3(b)) results in the inode being duplicated. Ext3cow allocates new inode 86 to which it copies the contents of inode 17. Inode 86 is assigned epoch 6 and marked as unchangeable. Inode 17 is brought to the current epoch and remains a live, writable inode.

2.4.2 Scoping Directory Entries

Directory entries are long lived, with a single name spanning many different versions of a file, each represented by a single inode. Figure 3 shows directory entries as a name-inode pair with the birth and death epoch as subscripts. The inode field points to the most recent inode to which the name applies. For example, name A points to inode 17 at the head of the inode chain. The name first occurred during epoch 5 and is currently live, represented by *. An * leaves live names open-ended so that as time progresses and the inode epoch increases, the directory entry remains valid. When removing a name, ext3cow updates the death epoch to indicate the point-in-time at which the name was removed. In Figure 3(c), name B dies and the death epoch is set to 8. The name B is no longer visible in the present and will not be visible for any point-in-time request that scopes to snapshot epoch 8.

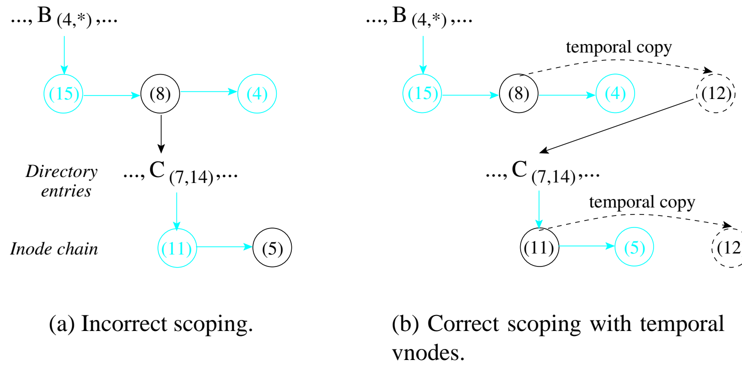
The flexibility of birth/death epoch scoping respects the separation between names and inodes in UNIX-like file systems. Many names may link to a single inode. Also, a different number of names may link to an inode during different epochs. The same name may appear multiple times in the same directory, linking to different inodes during non-overlapping birth/death periods.

2.4.3 Temporal Vnodes

The piece-wise traversal of file system paths makes it difficult to inherit time scope along pathnames. For paths of the form `.../B@time/C...`, time-shifting specifies that B, and its successors, are accessed at `time`. When accessing C, the file system provides only B's inode as context. Because `time` rarely matches the epoch number of B exactly, B's inode frequently has an epoch number prior to `time`. In this case, the exact scope is lost. For example, Figure 4(a) illustrates the wrong version of C being accessed. The access to C should resolve to the inode at epoch 11, but leads mistakenly to the inode at epoch 5.

To address this problem, ext3cow gives to each time context that accesses an inode a private in-memory inode (vnode) scoped exactly to the requested time. We call this a *temporal vnode* for two reasons: it is temporary and it implements time scoping inheritance. To make a temporal vnode, ext3cow creates an in-memory copy of the vnode to which the request scopes and sets the epoch number of the vnode to the requested time. It also changes the inode number to disambiguate the temporal vnode from the active vnode and other temporal vnodes. To avoid conflicts, the modified inode number lies outside of the range of inodes used by the file system. The temporal vnode correctly scopes accesses to directory entries (Figure 4(b)). This creates potentially many in-memory copies of the same inode data. Because data in the past are read-only, the copies do not present a consistency problem. The temporal vnode exists until the VFS evicts it from cache. Subsequent accesses to the same name (*e.g.* `B@12`) locates its temporal vnode in cache. Temporal vnodes are unchangeable and cannot be marked dirty.

Live inodes operate normally; concurrent or subsequent accesses in the present share a single copy of the vnode with the original inode number, corresponding to the inode on disk.



(a) Incorrect scoping.

(b) Correct scoping with temporal vnodes.

Figure 4: Accessing a path `...B@12/C...` in ext3cow. Directory entries are shown with birth and death epochs. Inodes (circles) are show with the epoch in which the inode was created. Inode numbers are not shown. Black directory entries and inodes indicate the access path according to scoping rules. The inode chain is traversed until an inode with creation epoch prior to the epoch of the parent inode is found. Temporal vnodes, in-memory copies of inodes, make this process accurate by preserving epoch information along access paths.

2.5 Performance Evaluation

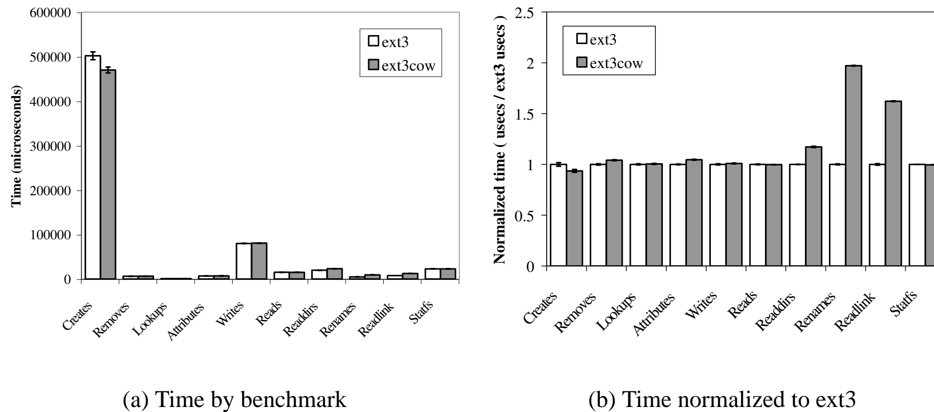
In order to quantify the cost/benefit trade-offs of versioning, we administered a variety of experiments comparing ext3cow to its sister file system – unmodified ext3. Experiments were conducted on an IBM x330 series server, running RedHat Linux 7.3 with the 2.4.19 SMP kernel. The machine is outfitted with dual 1.3 GHz Pentium III processors, 1.25 GB of RAM, and an IBM Ultra2 18.2G, 10K RPM SCSI drive. Experiments for both ext3cow and ext3 were performed on the same 5.8 GB partition.

2.5.1 Micro-benchmarks

The Connectathon NFS test suite evaluates operational correctness and measures performance. There are nine parts to the "basic" series of tests. Each part tests a separate system call. In order, they are: (1) create 155 files 62 directories 5 levels deep, (2) remove these files, (3) 150 getcwd calls, (4) 1000 chmods and stats, (5) write a 1048576 byte file 10 times and read it 10 times, (6) create and read 200 files in a directory using `readdir`, (7) create ten files, rename and stat both the new and old names, (8) create and read 10 symlinks, and, lastly, (9) perform 1500 statfs calls.

The results of the Connectathon basic test average the results of 20 runs on a newly mounted (cold cache) file system. Ext3cow meets the performance of ext3 in most areas. We present the average cumulative time to perform each test as bar graphs in both absolute time values (Figure 5(a)) and time normalized to the performance of ext3 (Figure 5(b)). Graphs include 95% confidence intervals.

Ext3cow and ext3 perform equally on tests that read inodes and data. Examples include tests 3 (Lookups), 5b (Reads), and 9 (Statfs). On these tests, the file systems execute the same code paths and manipulate the same data structures. Ext3cow also matches the performance of ext3 when writing and deallocating inodes. Tests 1 (Creates), 2 (Removes), and Test 4 (Attributes) show equivalence. Benchmark results indicate that ext3cow and ext3 are comparable when writing data (Test 5a, Writes). In practice, we expect ext3cow to incur a minor penalty on writes due to copy-on-write. String operations to support versioning result in



(a) Time by benchmark (b) Time normalized to ext3

Figure 5: Results from the "basic" tests in the Connectathon benchmark suite. All data are shown with 95% confidence intervals.

File System	Allocated Blocks	Allocated Inodes	Dir Inodes
ext3	1684696	1243263	15318
ext3cow – none	1684696	1243263	15318
ext3cow – 24 hour	1748126 (+3.8%)	1253642 (+0.1%)	33447 (+218%)
ext3cow – 1 hour	1850189 (+9.8%)	1289513 (+3.7%)	35440 (+231%)
ext3cow – 1 min	2144663 (+27.3%)	1370547 (+10.2%)	64458 (+421%)

Table 2: The total number of allocated inodes and the number of those inodes allocated for directories for the ext3 and ext3cow file systems over various snapshot frequencies.

ext3cow under-performing ext3 on tests dominated by name operations. During lookup, ext3cow parses every name looking for the @ version specifier. It performs similar string parsing when reading symbolic links. Tests 7 (Renames) and 8 (Readlinks) show string manipulation overhead. Test 3 (Lookups) does not have this overhead, because it does not call the on-disk file system lookup. Rather, test 3 calls the VFS entry point `getcwd`, which can be satisfied out of the VFS's directory entry cache. Test 6 (Readdirs) shows the overhead of scoping names in directories. The system does not parse strings or interpret the @ symbol during this test. The overhead comes from directory entry scoping only; ext3cow examines the birth and death epoch of every record that it reads. In total, micro-benchmark results indicate that ext3cow performs comparably to ext3 on data and inode operations and slightly worse on name operations.

2.6 Trace-driven Experiments

To examine the effect of snapshot on metadata allocation, we used four months of system call traces from Berkeley [48] to populate a file system partition and performed an off-line analysis to identify the type and amount of allocation. By aging a file system, we more accurately measure and analyze real-world performance [55]. The traces were played back through two file systems: ext3, as a baseline for comparison, and ext3cow. In ext3cow, we used three policies to quantify the allocation difference for various snapshot frequencies. Snapshots were taken at 24 hour, 1 hour and 1 minute intervals.

Table 2 displays a 0.1% increase in metadata for 24 hour snapshots and a 10.2% increase for 1 minute snapshots. These results indicate a small initial jump in the amount of metadata to support any amount of versioning, followed by gradual growth as snapshot frequencies increase. These results are consistent with those presented in CVFS [56]. Results show the storage cost of indefinite versioning to be quite small for snapshot intervals of an hour or more. Shorter snapshots (1 minute) produce larger overheads, although the storage requirements only increase by 27% over four months. Directory inode overheads are much greater, however, percentage overhead is not the right measure here. The total number of directory inodes is small when compared with all allocated inodes; they make up only 4.7% of all allocated inodes in the one hour snapshot trial and fewer that 3% in all others experiments. Thus, they have a small overall effect on the system.

2.7 Final Thoughts and Goals

Ext3cow is a fully implemented open-source file system that supports traditional applications of versioning: easy access to on-line backups; recovery from system tampering; read-only, point-in-time snapshots for data mining; and, file-oriented deletion recovery. Its performance is negligibly affected by versioning when compared with ext3. A paper on the technical implementation of ext3cow is to appear in the May 2005 edition of the ACM Transactions on Storage [42]. As it stands, ext3cow meets the mandated versioning and auditability requirements. However, ext3cow also continues to be a useful foundation for exploring technical solutions to other regulatory problems. In the following sections, we will explore two such problems: limiting liability through secure deletion and authentication in versioning systems.

3 Current Research - Secure Deletion

Versioning storage systems are increasingly important in research and commercial storage systems. However, existing versioning storage systems overlook fine-grained, secure deletion as an essential requirement. Secure deletion is the act of removing digital information from a storage system so that it can never be recovered. Fine-grained refers to removing individual files or versions of file, while preserving all other data in the system.

Secure deletion is valuable to security conscious users and organizations that wish to limit their liability in the regulatory environment. It protects the privacy of user data and prevents the discovery of information on retired or sold computers. Similarly, by securely deleting data after they have fallen out of regulatory scope, *e.g.* seven years for corporate financial records in Sarbanes-Oxley, data cannot be recovered even if disk drives are produced and encryption keys revealed. Data are gone forever and corporations are not subject to exposure via subpoena or malicious attack.

Currently, there are no efficient methods for fine-grained secure deletion in versioning storage systems. We believe the reticence to adopt secure deletion is founded in the lack of a practical implementation. Many methods for securely deleting data, such as the pulverization or large scale degaussing of a disk, are too coarse grained and complicated to be useful. The preferred and accepted methods for secure deletion in non-versioning systems include: overwriting data with other data such that the original data may not be recovered, called *secure overwriting* [21]; and, encrypting a file with a key and securely disposing of the key to make the data unrecoverable, called *key disposal* [8]. However, these techniques cannot by directly applied to versioning systems.

Secure overwriting has performance concerns in versioning systems. In order to limit storage overhead, versioning systems share blocks of data between file versions (Section 2.3). Securely overwriting a shared

block in a past version could erase it from subsequent versions. To address this, a system would need to detect data sharing dependencies among all versions before committing to a deletion. Also, in order for secure overwriting to be efficient, the data to be removed should be contiguous on disk. Non-contiguous data blocks require many seeks by the disk head – the most costly disk drive operation. By their very nature, versioning systems are unable to keep the blocks of a file contiguous in all versions.

Block sharing hinders key management in encrypting systems using key disposal. If a system were to use an encryption key per version, that key could not be discarded, as it is needed to decrypt shared blocks in future versions. To realize fine-grained secure deletion by key disposal, a system must keep a key for every shared block. Maintaining keys for every block is onerous and performance prohibitive.

We have developed two methods for the secure deletion of individual versions that minimizes the amount of secure overwriting while providing authenticated encryption. Our techniques combine disk encryption with secure overwriting so that a large amount of file data (any block size) are deleted by overwriting a small *stub*, typically 128 bits. For 4K, blocks, this is a 256 times speedup. Further, we collect and store stubs contiguously in a file system block so that overwriting a 4K block of stubs deletes the corresponding 1 MB of file data, even when file data are non-contiguous. Unlike encryption keys, stubs are not secret and may be stored on disk.

3.1 Related Work

Garfinkel and Shelat [18] give a survey of methods to destroy digital data. They identify secure deletion as a serious and pressing problem in a society that has a high turn-over in technology. They cite an increase in law suits and news reports on unauthorized disclosures, which they attribute to a poor understanding of data longevity and a lack of secure deletion tools. They identify two methods of secure deletion that leave disk drives in a usable condition: secure overwriting and encryption.

In secure overwriting, new data are written over old data so that the old data are irrecoverable. Gutmann [21] gives a technique that takes 22 synchronous passes over the data in order to degauss the magnetic media, making the data safe from magnetic force microscopy. (Fewer passes may be adequate [18]). This has been implemented in user-space tools and in a Linux file system [3]. Secure overwriting has also been applied in semantically-smart disk systems [54]. However, a large number of synchronous passes may be prohibitively expensive. Particularly for versioning systems that fragment file data.

For file systems that encrypt data on disk, data may be securely deleted by "throwing away" the corresponding encryption key [8]; without a key, data may never be decrypted and read again. This method works in systems that maintain an encryption key per file and do not share data between multiple files, unlike versioning systems and content-sharing stores [45]. This method greatly reduces the time needed to delete large amounts of data. The actual disposal of the encryption key often involves secure overwriting.

It should be noted that both secure overwriting and encryption are only effective when implemented in the file system. While user-space tools for secure deletion exist, these tools leak information because they are unable to delete metadata managed by a file system. Further, they can't be interposed between file operations, may leak actual data on truncates, and are difficult to use synchronously.

3.2 Secure Deletion in a Versioning File System

Secure deletion with versions builds upon authenticated encryption of data on disk. We use a keyed transform:

$$f_k(B_i, N) \rightarrow C_i || s_i$$

Input: Data $d_1,\ldots,d_m$, Block ID id, Counter x,
Encryption key K, MAC key M
1: $ctr_1 \leftarrow id||x||1||0^{128-|x|-|id|-1}$
2: $c_1,\ldots,c_m \leftarrow \text{AES-CTR}_K^{ctr_1}(d_1,\ldots,d_m)$
3: $t \leftarrow \text{HMAC-SHA-1}_M(c_1,\ldots,c_m)$
4: $ctr_2 \leftarrow id||x||0||0^{128-|x|-|id|-1}$
5: $x_1,\ldots,x_m \leftarrow \text{AES-CTR}_t^{ctr_2}(c_1,\ldots,c_m)$
6: $x_0 \leftarrow x_1 \oplus \ldots \oplus x_m \oplus t$
Output: Stub x_0, Ciphertext $x_1,\ldots,x_m$

(a) Secure deletion using AON encryption

Input: Data $d_1,\ldots,d_m$, Block ID id, Counter x,
Encryption key K, MAC key M
1: $k \xleftarrow{R} \mathcal{K}_{AE}$
2: $nonce \leftarrow id||x$
3: $c_1,\ldots,c_n \leftarrow \text{AE}_k^{nonce}(d_1,\ldots,d_m)$
4: $ctr \leftarrow id||x||0^{128-|x|-|id|}$
5: $c_0 \leftarrow \text{AES-CTR}_K^{ctr}(k)$
6: $t \leftarrow \text{HMAC-SHA-1}_M(ctr, c_0)$
Output: Stub $c_0, t, c_{m+1},\ldots,c_n$, Ciphertext $c_1,\ldots,c_m$

(b) Secure deletion using random key encryption

Figure 6: Two algorithms for authenticated encryption and secure deletion in versioning file systems.

that takes a data block (B_i), a key (k) and a nonce (N) and creates an output that can be partitioned into a secure data block (C_i), where $|B_i| = |C_i|$, and a short *stub* (s_i), whose length is a parameter of the scheme's security. In practice, s_i might be 128 bits. When the key (k) remains private, the transform acts as a secure authenticated encryption algorithm [6]. To securely delete an entire block, only the stub needs to be securely overwritten. This holds *even if the adversary is later given the key (k), e.g.* by subpoena. The stub reveals nothing about the data, and, thus, stubs may be stored on the same disk. A concept similar to stub deletion has been used in memory systems [16].

We present and compare two implementations of the keyed transform: one inspired by the all-or-nothing transform [9,46], the other based on randomized keys. We also present extensions, based on key-sharing, that allow for the control and deletion of data by multiple parties.

3.2.1 AON Secure Deletion

The all-or-nothing (AON) transform [9,46] ensures that an entire ciphertext, in our case a single file system block, must be decrypted before even one message block, some subset of the block, is revealed; no subset may be decrypted in isolation. The original intention of the AON transform was to increase the amount of time of a brute-force key search by a factor equal to the number of blocks in a ciphertext.

The all-or-nothing transform is the most natural construct for the secure deletion of versions. Our AON algorithm is presented in Figure 6(a). The algorithm takes a single file system block ($d_1,\ldots,d_m$), and performs encryption (Step 2) with a single file key, greatly easing key management. The encrypted data is authenticated (Step 3) and the result is then used to re-encrypt the data (Step 5). The resulting stub (Step 6) is not secret, rather, it is an expansion of the AON encrypted data.

AON encryption also enables the deletion of a block of data from an entire version chain. Due to the all-or-nothing properties of AON encryption, the secure overwriting of any 128 bits of a block will result in that block being securely deleted. This is the preferred technique for removing an entire version chain, as many blocks are shared between versions.

Despite these virtues, AON suffers from a known plain-text attack. After an encryption key has been revealed, if an attacker can guess the exact contents of a block of data, the attacker can verify that the data were once in the file system. Once the key is revealed, the attacker has all of the inputs to the encryption algorithm and may reproduce the ciphertext. The ciphertext may be compared to the undeleted block of

data, minus the deleted stub, to prove the existence of the data.

Such a plain-text attack is reasonable within the threat model of regulatory storage; a key may be subpoenaed in order to show that the file system contained specific data at some time. For example, to show that a doctor had knowledge of a patient's drug allergy in a malpractice case regarding mis-prescribed drugs.

3.2.2 Secure Deletion Based on Randomized Keys

To avoid such a plain text attack, systems must employ randomization, on a per-block basis, so that the encryption process is not repeatable. An algorithm for random-key secure deletion is shown in Figure 6(b). The scheme generates a random key, k, in Step 1 that is used to authenticate and encrypt a data block. To avoid the complexities of key distribution, we keep a single key per file, K, as with AON encryption, and use this key to encrypted the random key (Step 5). The encrypted randomly-generated key, c_0, serves as the stub. The expansion created by the AE scheme in Step 3 ($c_{m+1}, \ldots, c_n$) and the authentication of the encrypted random key (t) need not be securely overwritten to permanently destroy data. The encryption and storage of keys resembles lock-boxes in the Plutus file system [29].

The algorithm is built upon any Authenticated Encryption (AE) scheme (Step 3). This algorithm is provably secure when the underlying AE scheme is secure; AES and SHA-1 satisfy standard security definitions. An advantage of this transform is its speed. For example, when the underlying AE is OCB [47], only one pass over the data is made and it is fully parallelizable.

Randomized key encryption does not hold all the advantages of an AON scheme. Only selective components may be deleted, *i.e.* c_0. Thus, in order to delete a block from all versions, the system must securely overwrite all stub occurrences in a version chain, as opposed to securely overwriting only 128 bits of a data block in an AON scheme. Additionally, the algorithm suffers from a larger message expansion: 384 bits per disk block are required instead of 128 required for the AON scheme. We are exploring other more space-efficient algorithms.

3.2.3 Secure Deletion with Secret-Sharing

Our random-key encryption scheme allows for the separation of the randomly-generated encryption key into key shares. Any number of randomly generated keys may be created in Step 1 (Figure 6(b)). and composed to create a single encryption key, k. With key shares, any single share may be destroyed to securely delete the corresponding data. However, all key shares must be present at the time of decryption. For example, a patient may hold a key share for their medical records on a smart-card, enabling them to control access to their records, and also independently destroy their records without access to the storage system.

3.3 Final Thoughts and Goals

Our present contributions include two algorithms for the efficient secure deletion in storage systems that share data between files, in particular, versioning file systems that comply with federal regulations. We are in the midst of secure deletion research and development, and expect preliminary performance results in the coming months. We plan to compare the performance of various secure deletion techniques with secure stub deletion. This includes measuring various cryptographic algorithm throughput, investigating security versus performance trade-offs for overwriting techniques, and comparing various stub block placement strategies.

4 Future Research Vision and Goals - Constructing a Verifiable Audit Trail

The principal goal of this project is to construct an audit trail for a versioning file system so that the changes made to data, and the order in which they occurred, may be verified. This builds upon ext3cow's versioning model in which all versions of a file may be accessed at any point-in-time. Thus, data are never deleted, files are updated in such a way that the new and old versions are both accessible. Over time, many versions of a file accrete.

The audit trail created allows an auditor to positively confirm the contents of the file system at a particular point in the past. The audit process uses authentication data stored at a trusted third party and requests the file system to produce data (from the past) consistent with the escrowed authentication data. If the file system cannot produce consistent data, it fails the audit, indicating that data were tampered with, corrupted, or destroyed. If the file system can produce consistent data, this will provide a strong guarantee of the data's authenticity; *i.e* if the file system produces consistent data, these data are exactly the data that were stored in the past and used to generate the authentication data.

The audit trail and verification process supports the compliance models used in federal legislation for record retention. For example, in Sarbanes-Oxley, corporations and their accountants are required to retain financial records independently. An auditor may compare such records for consistency at a later date. If the records do not match, the audit fails and the corporation-accountants pair have failed to comply with legislation. Similarly, our system provides a positive statement, verifying the retention of past data in compliance with regulations. Conversely, the system does not prevent the destruction of data. A failed audit means that the file system does not have the exact data stored in the past. It does not allow those data to be reconstructed. The illegal destruction of documents provides an accurate analogy in the physical world. Failure to produce those documents during audit is a compliance failure. It is not a proof of wrongdoing or maliciousness. However, legislation includes substantial penalties for lack of compliance, both for electronic and paper records, that make audit trails and verification meaningful.

Finally, we will require the generation of audit information to be computationally efficient. This issue has many dimensions. Primarily, the effort to compute the audit trail should scale with the number of blocks being written, as opposed to the size of the data. This matches the I/O efficiency of versioning file systems, which share data between versions, writing only modified or added data to disk through the copy-on-write process. We also would like to minimize the amount of information in the audit trail and the frequency with which the audit trail needs to be updated. Given that file systems may create thousands of versions a second and contain millions of files, it would seem necessary to avoid sending authentication data for every file on every update. Towards this end, we will explore aggregation and summarization. In aggregation, authentication data may represent many file system elements, such as collections of files, directories, or subtrees of the file system namespace. In summarization, authentication data may represent multiple versions of the same file system element.

4.1 Description

For a file, an audit trail will include verification and authentication of individual versions as well as the version sequence – equivalent to a log of the changes to a file over time. During the creation and modification of a file version, the system produces a small amount of authentication data to be escrowed at a trusted third party. We call this data a *version authenticator*. At a later time, an auditor may request that the file system produce file data consistent with the escrowed version authenticator. This audit process will verify that the file system stored and continues to store that data. In addition to authenticating the data of an individual file version, the auditor will be able to verify a the unique sequence of versions leading up to a file. This is

accomplished by using two version authenticators from two different points in time. To verify a sequence of versions, the file system reconstructs all file versions between those two points in time and the auditor verifies that this sequence transforms the first authenticator into the second.

The escrow process will incur little in the way of storage and bandwidth overheads. File data and version sequences may be verified on a small amount of authentication data escrow infrequently a secure message authentication (256 bits) will suffice. 256 bits is small percentage of the average file size, and so storage overhead, even for extremely large systems, will be insignificant; 15 gigabytes for a one billion file system. Additionally, authenticators may be transmitted infrequently, as a version authenticator represents the data for that version and all preceding versions, when considered in conjunction with a previous version authenticator.

Outstanding technical challenges in this project include the construction of authenticators for complex file system structures and the efficient verification of data during audit. Currently, we have insight and methods into the construction of an authenticator for a sequence of file versions. These methods do not extend trivially to directories of files and directory hierarchies. It is natural to hierarchically extend the authentication scheme, so that the authenticator for a directory is secure composition of the version authenticators for its files and subdirectories. Continuing this process, we could create a single authenticator at the root of the file system that represents all data, metadata, and directories in the entire system. This is attractive because it minimizes the amount of data to be escrowed. However, it has the drawback of requiring the auditor to access all data in the entire file system at a point in time to verify the authenticator, which is infeasible for large data systems. We observe that there is a natural trade-off between the granularity and frequency of authentication data and the computational and I/O effort in verification. The structure of this trade-off and some potential alternatives are discussed in Section 4.4.

For now, we will restrict our discussion to the generation and escrow of authenticators for version histories of individual files.

4.2 Threat Model

Audit trails serve to verify that the data present in the system at point in time t_a is the same as the data used to generate the authentication data at time $t_g : t_g < t_a$. In the basic audit model, there are three participants:

1. The file system *owner* (O) has complete control over the file system. The owner may read and write file system data consistent with compliance requirements. They may also act out of compliance; they may delete, rewrite, or destroy data, and they may do so either through file system interfaces, accessing the disk through driver interfaces, or by physically access to the disk.

2. The *escrow agent* (EA) is a third party trusted by both the auditor and the owner. The agent receives, stores, and reproduces version authenticators. It also binds authenticators with a time.Once stored and bound, the agent will not modify or delete a version authenticator. The EA responds to requests for a version authenticator from any point-in-time with the authenticator written at that time.

3. The *auditor* (AU) has no control over either the file system or the escrow agent's store. It makes requests of EA for version authenticators and then challenges O to produce version histories consistent with the version authenticator it receives. The auditor decides whether the owner is in compliance or not.

The principal attack against which this system defends is the creation of false version histories that pass the audit process. This includes protecting against this attack by the owner, which has access to all the information used to generate version authenticators. The attach scenario proceeds as follow:

1. O and AU have an agreed upon algorithm for the generation of version authenticators.

2. At times $t_i | i \in [1,n), t_i - 1 < t_i < t_i + 1$, O creates versions v_i.

3. On the creation of v_1 at time t_1, the owner transmits the corresponding version authenticator A_{v_1} to EA.

4. On the creation of v_n at time t_n, the owner transmits the corresponding version authenticator A_{v_n} to EA.

5. At time $t_a > t_n$, AU requests and receives A_{v_1} and A_{v_n} from EA.

6. AU then requests $v_1, \ldots, v_n$ from O and receives a sequence of versions $r_1, \ldots, r_m$

7. AU also requests and receives from O any encryption keys or other secrets used in the generation of A_{v_1} and A_{v_n}.

O successfully attacks AU, whenever the version sequences are not identical $r_1, \ldots, r_m \neq v_1, \ldots, v_m$ and AU cannot distinguish that they are different based on A_{v_1} and A_{v_n}.

The false version history attack encompasses many variants. As part of an audit, the auditor may have the power to compel O to give AU access to the file system. However, an O that can perform the false version history attack can write that false version history to the disk so that AU could not distinguish the disk contents from the original history.

Any attacks against the EA also threaten the system. A malicious attacker M could modify the version authenticators stored at EA so that even if O presents a correct version history, it fails the audit. Also, O could attach EA to modify a version authenticator consistent with changes it has made to the file system, allowing it to pass an audit on a false history. However, we treat EA as a trusted third party and, thus, do not address attacks against EA.

4.3 Security Constructs

We employ a parallel message authentication code (PMAC) [4, 5, 7] for version authenticators in order to achieve I/O efficiency. A version of a file shares data with its predecessor; it differs only in the blocks of data that are changed. As a consequence, the file system performs I/O only on these changed blocks. By using the PMAC, we create the authenticator for the new version using the authenticator of the predecessor and the data of the changed blocks. We say that the authenticator is *incrementally calculable*. In this way, the computational effort to compute the authenticator scales with the amount of I/O performed, rather than the file size. In contrast, a serial hash MAC, such as SHA-1, would require the whole file to be examined in the construction of the MAC.

The input to a version authenticator also includes the authenticator to the previous version in order to bind each authenticator to a unique version history. Specifically, we take a file v_i consisting of blocks $v_i = \{b_0, \ldots, b_n\}$ each of size B and construct the version authenticator (defined iteratively) as:

$$A_{v_i} = \text{PMAC}(v_i || A_{v_{i-1}}); i \in (1, \infty), A_0 = 0. \tag{1}$$

In this way, a version authenticator authenticates the content of the file version as well as binds it to a previous version. By induction, the allows the auditor AU to verify the entire version history between v_1 and v_n based upon A_{v_1} and A_{v_n}.

We use the parallel property of the PMAC to perform computations separated in time, rather than the original intended use of separating computation in space. PMAC computes a one-way function on each block of the input. To be near consistent with the original publication, for block b_i, we label the one-way function $Y(b_i)$. The output of the PMAC is the exclusive-or of all the input blocks: In our case

$$A_{v_i} = \bigotimes_{j=0}^{n} Y(b_j) \otimes Y(A_{v_{i-1}}). \tag{2}$$

This form is the full computation. There is also an incremental computation. Assuming that the next version v_{i+1} differs from v_i only in block b_k, which has been changed to b'_k. We observe that

$$A_{v_{i+1}} = A_{v_i} \otimes Y(b_k) \otimes Y(b'_k) \otimes Y(A_{v_{i-1}}) \otimes Y(A_{v_i}), \tag{3}$$

is the incremental form of the computation. This extends trivially to any number of changed blocks. The updated version authenticator adds the contribution of the changed blocks and removes the contribution of those blocks in the previous version. Similarly, it updates the information that links the previous version ($A_{v_{i-1}} \rightarrow A_{v_i}$). Thus, the computation of version authenticators scales with the amount of I/O done between versions.

More importantly, the computation of the updated version authenticator can be performed on data available in the cache, requiring no additional disk I/O. The changed data is, by definition, written to cache and was in cache prior to being written. $Y(A_{v_i})$ can be generated from the changed data. In practice, nearly all cryptography is I/O bound, which makes avoiding additional I/O quite significant.

4.4 Open Challenges

We are concerned with the efficiency of conducting an audit against a large body of data. The construct we use for a version authenticator requires all the data of a file to be accessed during audit. For files this task is reasonable in that files are minimum management entities, *i.e.* in a file system, it does not make sense to audit only a portion of a file. Similarly, to verify a version chain, all data in all versions in the chain must be accessed during audit. For version chains, this creates a trade-off between the frequency of sending authenticators to the third party and the effort to verifying version histories. For example, if an authenticator were escrowed for every version, then the system accesses only the desired versions. For authenticators separated by many versions, no subsets of the intermediate version may be evaluated without constructing the history between the authenticated versions.

Aggregating the version authenticators of files could provide efficiencies in the construction of authenticators and the transfer of data to and storage of data at the third party. However, for obvious implementations, aggregation is detrimental to the audit process. As an example, one could construct an aggregate authenticator for a directory, formed as a PMAC of the version authenticators for individual files. The aggregate more compactly authenticates all of the files, but requires that all data from all files be accessed to verify the aggregate.

In the coming months, we plan to implement version authentication in the ext3cow file system. We expect a working implementation by the end of Summer 2005, and a full set of results by the Winter of 2005. Barring unforeseen circumstances, this body of regulatory storage work will be finalized in the Spring of 2006 as a dissertation, submitted in partial satisfaction of the requirements for the degree of Ph.D. in Computer Science.

References

[1] A. Azagury, M. E. Factor, and J. Satran. Point-in-time copy: Yesterday, today and tomorrow. In *Proceedings of the Tenth Goddard Conference on Mass Storage Systems and Technologies*, pages 259–270, April 2002.

[2] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[3] S. Bauer and N. B. Priyantha. Secure data deletion for Linux file systems. In *Proceedings of the USENIX Security Symposium*, 2001.

[4] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the 27th ACM Symposium on the Theory of Computing*, pages 45–56, 1995.

[5] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In D. Coppersmith, editor, *Advances in Cryptology - Crypto 95 Proceedings, Lecture Notes in Computer Science*, volume 963, pages 15–28. Springer-Verlag, 1995.

[6] M. Bellare and C. Namprempre. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology - Asiacrypt 2000 Proceedings, Lecture Notes in Computer Science*, volume 1976. Springer-Verlag, 2000.

[7] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - Eurocrypt 2002 Proceedings, Lecture Notes in Computer Science*, volume 2332, pages 384 – 397. Springer-Verlag, 2002.

[8] D. Boneh and R. Lipton. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium*, pages 91–96, July 1996.

[9] V. Boyko. On the security properties of OAEP as an all-or-nothing transform. In *Proceedings of CRYPTO '99*. Springer-Verlag, 1999.

[10] R. Bryant, R. Forester, and J. Hawkes. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 259–274, June 2002.

[11] R. Card, T. Y. Ts'o, and S. Tweedie. Design and implementation of the second extended file system. In *Proceedings of the 1994 Amsterdam Linux Conference*, 1994.

[12] A. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*, March 1998.

[13] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992.

[14] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 19–28, June 2004.

[15] Digital Equipment Corporation. *Vax/VMS System Software Handbook*, 1985.

[16] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to forget a secret. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, 1999. Lecture Notes in Computer Science, 1563, Springer-Verlag.

[17] EMC Corporation. *EMC TimeFinder Product Description Guide*, 1998.

[18] S. L. Garfinkel and A. Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy*, 1(1):17–27, 2003.

[19] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

[20] R. J. Green, A. C. Baird, and J. Christopher. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, 8(2):32–45, 1996.

[21] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–90, July 1996.

[22] J. Hagerty. Sarbanes-Oxley compliance spending will exceed $5b in 2004. *AMR Research Outlook*, Dec 2004.

[23] R. Hagman. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating systems principles (SOSP)*, pages 155–162, 1987.

[24] Hitachi, Ltd. *Hitachi ShadowImage*, June 2001.

[25] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the USENIX San Francisco 1994 Winter Conference*, January 1994.

[26] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51 – 81, February 1988.

[27] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *3rd Symposium on Operating System Design and Implementation Proceedings (OSDI)*, pages 239–250, February 1999.

[28] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, 6(1):51–81, 1996.

[29] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, March 2003.

[30] P. Killbridge. The cost of HIPAA compliance. *New England Journal of Medicine*, 348(15):1423–1424, 2003.

[31] S. R. Kleiman. Vnodes: An architecture for multiple file system in SUN UNIX. In *Proceedings of the 1986 USENIX Summer Technical Conference*, pages 238–247, 1986.

[32] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[33] M. K. McKusick. Running "fsck" in the background. In *Proceedings of the BSDCon 2002 Conference*, pages 55–64, February 2002.

[34] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[35] M. K. McKusick and G. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Freenix Track at the Annual USENIX Technical Conference*, pages 1–17, June 1999.

[36] M. K. McKusick, W. N. Joy, J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[37] Sun Microsystems. *NFS: Network file system protocol specification*. Network Working Group, Request for Comments (RFC 1094), March 1989. Version 2.

[38] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[39] L. Moses. An introductory guide to TOPS-20. Technical Report TM-82-22, USC/Information Sciences Institute, 1982.

[40] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 115–128, 2004.

[41] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 117–129, Jan 2002.

[42] Z. N. J. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transcations on Storage*, 1(2), 2005.

[43] D. Presotto. Plan 9. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 31–38, April 1992.

[44] S. Quinlan. A cached worm file system. *Software – Practice and Experience*, 21(12):1289–1299, Dec 1991.

[45] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File And Storage Technologies (FAST)*, pages 89–101, January 2002.

[46] R. L. Rivest. All-or-nothing encryption and the package transform. In *Proceedings of the 1997 Fast Software Encryption Conference*, pages 210–218, 1997. Springer Lecture Notes in Computer Science #1267.

[47] P. Rogaway, M. Ballare, J. Black, and T. Krovet. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM Conference on Computer and Communications Security*, pages 196–205, 2001.

[48] D. Roselli and T. E. Anderson. Characteristics of file system workloads. Research report, University of California, Berkeley, June 1996.

[49] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[50] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Summer USENIX Conference Proceedings*, pages 119–130, June 1985.

[51] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

[52] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, pages 25–34, 1985.

[53] M. Seltzer, K. Bostic, M. K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Conference, San Diego, CA, USA, 25-29 January 1993*, pages 307–326, January 1993.

[54] M. Sivathanu, L. Bairavasundatam, A. C. Arpaci-Dussaeu, and R. H. Arpaci-Dusseau. Life or Death at Block-Level. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, December 2004.

[55] K. A. Smith and M. I. Seltzer. File system aging – Increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS Conference*, pages 203–213, June 1997.

[56] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, March 2003.

[57] J. D. Strunk, M. L. Scheinholtz G. R. Goodson, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, October 2000.

[58] A. S. Tannenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1987.