TOWARDS REGULATORY COMPLIANT STORAGE SYSTEMS

by

Zachary Nathaniel Joseph Peterson

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2006

Abstract

Legislators have begun to recognize the importance of how electronically stored data should be maintained and secured. Similarly, the courts have begun to differentiate electronic data from their paper analogs. Examples of some sweeping 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.

Many current storage solutions fail to meet the new demands legislation placed 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.

We present three technical contributions to the field of regulatory compliant storage. The first is an open-source versioning file system designed to be a platform for developing regulatory compliant storage technologies. We then introduce algorithms and an architecture for the secure deletion of individual versions of a file. Lastly, we construct an audit trail model for a versioning file system so that the changes made to data, and the order in which they occurred, may be verifiable.

Advisor: Randal Burns
Readers: Randal Burns
Avi Rubin
Darrell Long

Acknowledgements

This dissertation is the culmination of six years of graduate school at two universities, on two different coasts, completed only with the support of many people. First and foremost, I deeply thank my adviser and good friend, Randal Burns, without whom this work, let alone any successes I've had as a scientist, would not have been possible. His support proved unwavering and his guidance unfaltering. Throughout our years of working together, Randal has been more akin to a brother than a boss; just as quick to come to my aid as to critique my mistakes. This was truly a unique relationship; one of family, rather than contract. I look forward to being his new peer and continued friend.

In addition to Randal, I was extremely fortunate to be exposed to a faculty of exceeding talent and energy. I had the distinct pleasure of working closely with two faculty members, Avi Rubin and Giuseppe Ateniese, both of whom provided me unique opportunities for expanding my research horizons. I am grateful to them both. Of all the faculty, however, Andreas Terzis was an uncommon counselor. Whether it be exchanging new ideas at the gym or pondering life in front of the Xbox, Andreas was always willing to lend a sympathetic ear and a critical mind. Thank you.

Research is rarely a solo endeavor and throughout my career at Hopkins I was priveleged to be able to work with a set of excellent peers. My colleagues in the Hopkins Storage Systems Lab: Alexandros Batsakis, Chuck Wu, Dan Wang, Eric Perlman and Tanu Malik, spent many hours of peer review to benefit me and my work, and for that, I am indebted. Of particular distinction, Alexandros showed me that it's possible to achieve the rare combination of cool and computer science. Outside of the HSSL, Adam Stubblefield, Joe Herring and Steve Bono provided significant insight into the worlds of cryptography and system design; the foundations of this work.

Special thanks go to Scott Banachowski, who has housed me on conference trips, reviewed my work *ad nauseam*, shared his love of music with me, and has consistently been a model of the scientist I aspire to be.

My parents have always been, and continue to be, the cornerstone of my endeavors. They have made everything possible. I hope to finally be able to provide them with the top-notch technical support they deserve. And to my brother, Andrew, who has shown me compassion, at all hours, and above all else.

And lastly, I dedicate this work to Jamie Funderburk, who was able to make her love and support known to me both at home and abroad. Her contributions to my life are without measure and will never be forgotten.

Contents

List of Tables

List of Figures

Chapter 1

Legislating Storage Systems

> "Good laws have their origins in bad morals."
>
> – *Ambrosius Macrobius*

The introduction of computers to the workplace has resulted in a record management paradigm shift. Paper records have made way for their electronic counterparts, allowing information to be quickly indexed and shared as well as improving accuracy. However, the rapid adoption of electronic records is not all good news; with new technology come new threats. Duplicates of records can be made and transmitted anywhere instantaneously, threatening privacy. Electronic data is also easily malleable, making undetectable forgery and falsification a simple task. Legislators and the courts have begun to recognize the importance of how electronic records need to be maintained and secured. The result is an ever increasing body of electronic record management

Recent federal, state and local legislation have created new requirements for how to retain and access electronic information. There currently exist over 4,000 acts and regulations that govern digital storage, all with a varying degree of requirements for maintaining electronic records. Examples 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.

The legislation does not mandate specific technologies, nor is it made obvious how a storage system should technically meet all of the requirements. Despite this confusion, storage systems vendors have quickly identified the large market opportunity and have modified existing systems and marketed them as compliance products. In 2004, Sarbanes-Oxley compliance alone represented a market of over $5 billion [48]. Mostly, vendors add policy enhancements to existing storage platforms that aid in the maintenance and retention of data, such as forbidding data deletion. However, many of these products fail to meet the true demands legislation places on storage systems [34, 55, 93]. They have been unable to combine the data retention, privacy and authenticity requirements into a single system. In this dissertation, we present some completed technical contributions to the field of regulatory compliant storage pursuant to this goal. We have implemented these solutions, and made them available for public use.

We begin this chapter by detailing some of the federal laws and regulations that affect computer storage system architectures. We distill the legislation into a set of four technological requirements. We conclude that compliance requires: versioning with real-time access to past versions, secure deletion, digital privacy and digital authenticity. By no means is our analysis of legislation exhaustive, but it does demonstrate the breadth and spirit of the legislated requirements, illustrate the similarities between the laws, and motivate the need for better technological solutions.

1.1 Health Insurance Portability and Accountability Act

The Health Insurance Portability and Accountability Act (HIPAA) [22], enacted in 1996, was written to develop standards for the normalization of individual health records and to encourage the use of electronic records pursuant to these goals. HIPAA requires "covered entities," including doctors, hospitals, insurance, billing and clearing house companies, to adhere to a set of resolutions designed to standardize electronic health care transmissions and protect the privacy and security of individually identifiable health information. As it relates to technology, HIPAA includes provisions that address the security and privacy of "protected health information" (PHI); specifically, two Administrative Simplification provisions, the Privacy Rule and the Security Rule. Both rules affect the way electronic records must be accessed, transmitted, and managed.

1.1.1 Privacy Rule

The HIPAA *Standards for Privacy of Individually Identifiable Health Information*, or Privacy Rule [23], is the first comprehensive set of Federal protection standards for the privacy of personal health information. The Rule addresses the use and disclosure of individuals' PHI by covered entities. The Rule attempts to strike a balance between the highest protection of privacy for individuals with the needs for smoothly flowing health care information. Additionally, The Rule attempts to be flexible enough to cover the large variety and sizes of health care providers. The Rule's key requirements include *access control* and *error correction* procedures, allowing an individual to manage how their personal information will be used, including limiting the marketing of their PHI. The Privacy Rule also requires a covered entity to document all of its privacy procedures and designate a privacy officer to implement the procedures and train staff.

1.1.2 Security Rule

The HIPAA Security rule acts as a complement to the Privacy Rule, providing guidance for interpreting the physical and technical safeguards required by the Privacy Rule. The Security Rule, however, only protects Electronic Protected Health Information (EPHI) and does not cover paper copies of documents or oral information. The Rule defines three segments of security safeguards for compliance: administrative, physical, and technical. Generally, The Rule requires a covered entity to ensure the *confidentiality*, *integrity* and *availability* of all EPHI that is created, received, maintained or transmitted by a covered entity. The entity must protect against reasonable threats and hazards, as well as protect against any reasonably anticipated misuse or unauthorized disclosure.

1.2 Sarbanes-Oxley Act

The Sarbanes-Oxley Act of 2002 (SOX) [21] is legislation enacted in response to the high-profile Enron and WorldCom financial scandals [65, 67]. Its grand scale and sweeping implications have made it the subject of much independent research [29, 33, 34, 55]. In its own words, SOX was designed to "protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws and for other purposes." SOX is administered by the Securities and Exchange Commission (SEC), charged to publish rules and set deadlines for compliance. The Act does little to set business practices, nor does it specify how a business should store its financial records, leaving many businesses in the dark as to which technologies are necessary to become compliant. In this way, SOX affects IT departments as strongly as the financial side of corporations. All business records, including electronic records and messages, must be retained for "not less than five

years." The records must be stored *authentically* and be available for *real-time reporting*. Consequences for non-compliance are steep fines, imprisonment, or both. This leaves IT departments with the challenge of creating and implementing controls and procedures in a cost-effective fashion that satisfy the requirements put forth by the legislation. We identify those sections of Sarbanes-Oxley that require a technological solution and particularly affect a company's storage infrastructure.

1.2.1 Section 302

Section 302 centers on the responsibilities of a company's management with respect to the electronic records they keep. CFOs and CEOs must personally certify that their financial records accurately represent the company's financial condition. Additionally, they certify to an audit committee or independent auditor that they are responsible for the proper disclosure controls and procedures, ensuring no deficiencies in control, data manipulation, or acts of fraud are committed in their financial records.

1.2.2 Section 404

Section 404 mandates an annual evaluation of internal controls for financial reporting. The report must address the record management procedures and the effectiveness and control structure of the company's financial records. In addition, the company's auditor must issue a report that attests to the effectiveness of the internal controls and procedures. An auditor must be able to verify the authenticity of the records to pass the auditing process.

1.2.3 Section 409

Section 409 addresses real-time reporting on financial records. The legislation requires that "[e]ach issuer... shall disclose to the public on a rapid and current basis such additional information concerning material changes in the financial condition or operations of the issuer". The results must be "in plain English."

1.2.4 Section 802

Section 802 requires companies to produce and maintain authentic and immutable records for at least five years. The types of records are specified by the SEC, and include:

> *records such as workpapers, documents that form the basis of an audit or review, memoranda, correspondence, communications, other documents, and records (including electronic records) which are created, sent, or received in connection with an audit or review and contain conclusions, opinions, analysis, or financial data relating to such an audit or review.*

A failure to store the required documents or knowingly altering, destroying or falsification of relevant data may result in fines and up to twenty years of imprisonment. Section 802 applies to both public and private companies under certain circumstances.

1.3 Gramm-Leach-Bliley Act

The Gramm-Leach-Bliley Act (GLBA) [24], also known as the Financial Services Modernization Act, increases competition among banks, securities companies and insurance companies by allowing investment and commercial banks to consolidate. In addition, legislators proposed best practices in consumer privacy, data and information protection, and securities regulation, maintaining that the privacy of consumer financial information is the primary concern. Specifically, three

sections: the Financial Privacy Rule, the Safeguards Rule, and Pretexting Protection. These rules govern the collection, disclosure, and protection of consumers nonpublic, personal information and personally identifiable information.

1.3.1 Financial Privacy Rule

This rule requires privacy notices to be issued by a financial institution when creating a new consumer relationship and annually thereafter or when the privacy policy is modified. The notice must detail what consumer information is collected, how the information is used, with whom the information is shared, and, most importantly, how the information is being protected. The notice must also provide an opt-out option for consumers who do not wish to share their private information.

1.3.2 Safeguards Rule

The Safeguards Rule mandates that financial institutions must develop a plan that addresses the security and protection of clients' private personal information, including former clients. The plan must include a strategy for developing and testing their architecture to secure personal information, as well as establishing a program to update safeguards commensurate with threat levels.

1.3.3 Pretexting Protection

GLBA requires that financial institutions take the necessary precautions to protect clients from pretexting or "social engineering" attacks. These types of attacks include "phishing" or other identity thefts that involve the fraudulent use of client information.

1.4 Federal Information Security Management Act

The E-Government Act was signed into law in December 2002 with Title III of the act being the Federal Information Security Management Act, or FISMA [25,28]. FISMA is a replacement of the Government Information Security Reform Act (GISRA) designed to broaden and strengthen computer and network security in both the federal government and its contractors. The goals of FISMA are to: provide a security control framework for electronic information, provide a set of minimum controls required for federal data, provide a framework for improved oversight, and recognize that specific hardware and software solution decisions should be left to individual agencies. Agencies are required to implement yearly risk assessments, tests and evaluations.

1.5 SEC Rule 17 CFR § 240.17a-4

The Security Exchange Commission (SEC) Rule 17 CFR § 240.17a-4 makes detailed requirements on electronic record management for brokers and members of the exchange [20,56]. The Rule addresses how and when electronic records should be stored on "electronic storage media." It stipulates that the media "[p]reserve the records exclusively in a non-rewritable, non-erasable format" as well as maintaining the records such that they are "accurately reproduced" and stored in an "unalterable form." Further, the storage device must "[v]erfiy automatically the quality and accuracy of the storage media recording process." Section 240.17a-4(f)(2)(ii)(C) makes a timestamping and serialization requirement intended to "ensure both the accuracy and accessibility of the records by indicating the order in which records are stored, thereby making specific records easier to locate and authenticating the storage process." Records must also be kept readily available for review at any time.

1.6 The Court and Electronic Records

The rules of the court have also been updated to account for the ephemeral and evolving nature of electronic data. Previously, courts required litigants to preserve and produce all electronic records in their "possession, custody, or control" that are relevant to the proceedings. However, a KCI Research report [93] states that because electronic evidence is becoming a routine aspect of litigation, courts are developing rules and requirements that highlight the need for companies to be vigilant in managing and producing all relevant evidence. The courts are uninterested in storage formats or media technologies. Data must be accessible, authentic and inexpensive to store in the context of a lawsuit. Simple backups and business continuity plans are not an acceptable substitute for dedicated archives that meet the systematic retention, preservation and accessibility requirements set forth by recent legislation. In one instance, a company was sanctioned by the court over the reuse of backup tapes containing relevant email messages, even though the company "did not do so willfully, maliciously, or in bad faith." In another, Author Anderson was cited for obstruction of justice in the Enron investigation for poorly managed and unauthentic electronic data records.

1.7 Distilling the Requirements

All of this legislation makes broad technological requirements of storage systems. However, many share similar goals, allowing a few technologies to encompass and satisfy their legislated mandates. We distill these requirements into four technologies: versioning with real-time access, secure deletion, digital privacy and digital authenticity.

1.7.1 Versioning with Real-Time Access

Legislation requires an auditable trail of changes made to electronic records that are accessible in real-time. SEC Rule 17 CFR §240.17a-4(f)(2)(ii)(C) mandates that a storage system "serialize" and "time-date" electronic records. Section 802 of SOX demands that electronic records be retained immutably; HIPAA and FISMA have similar immutably requirements. These requirements imply the need for versioning files over time. Each version must be an immutable record of how that file looked at a given point in time. Every modification to a file must create a new version labeled with a time stamp, giving an implicit order to the modifications. Versions of a file must be chained together, providing a complete modification history of the life of the file over time. In this way, the system creates a logical relationship between the versions of the file as well as a temporal relationship between other file versions in the system.

In addition to versioning, the system must provide a means of quickly retrieving versions from any point in time. This meets the "real-time access" requirements of SOX Section 409, SEC Rule 17 CFR §240.17a-4(f)(2)(ii)(D) and 17a-4(f)(3)(i), as well as the HIPAA Security Rule's "availability" requirement. Real-time access means having the ability to fetch any version of a file or directory from any point in time with the same speed as accessing the actively running file system. The file system must present an uniform interface for accessing past versions as well as providing a logical view into the past, *i.e.* being able to visualize previous system hierarchies. This allows an auditor, for example, to view past file system states as a whole, rather than a disjoint collection of versions.

1.7.2 Secure Deletion

Legislation requires that electronic data records have a limited scope. This includes protecting former clients' privacy (GLBA Safeguard Rule), limiting the length of time for which a company is liable for maintaining accurate financial records (SOX Section 802), and the right of a patient to redact portions of their medical records (HIPAA Privacy Rule). Destroying electronic records is a more challenging problem then destroying paper analogues. The physical proprieties of magnetic storage and the design of most file systems allow data to exist even after explicit deletion commands. Storage systems must use *secure deletion* techniques to meet compliance requirements. Secure deletion is the act of permanently removing data from a system, either by physically removing the data from the medium, or by making data unreadable. NIST has published a set of federal guidelines for securely removing data from obsolete forms of storage [103], however, the recommendations do not address deleting data from a live environment, *i.e.* securely removing data from the active file system without affecting other data. Deletion must also be fine grained. By fine grained, we mean the system must be able to securely delete subsets of a file on an active file system, as in the redaction of a patient's medical record.

1.7.3 Digital Privacy

Systems must now provide confidentiality through encrypted storage and data transmission. By using encryption, systems may meet the access control requirements of the HIPAA Privacy Rule. Only those users who possess the proper decryption keys will be able to access data in a meaningful way. Further, an accidental disclosure of encrypted data does little to threaten the privacy of patients. Other legislation, including the HIPAA Security Rule and GLBA Privacy Rule, explicitly

require the use encryption in data systems for consumer and patient privacy. FISMA also requires federal agencies to protect its data from unauthorized disclosure by using encryption commensurate with the sensitivity of the information.

1.7.4 Digital Authenticity

In addition to encryption, system must also employ authentication to meet legislated requirements. Authentication in a storage system provides three key features: data integrity, user authenticity, and authentic client-server transactions. The HIPAA Security Rule, Section 802 of SOX, and SEC Rule 17 CFR §240.17a-4(f)(2)(ii)(B) require a verification of the "accuracy" and "integrity" of electronic data. Additionally, Section 404 of SOX requires that auditors be provided with proof of the integrity of data. While encryption provides privacy from unauthorized intrusion and disclosure, it alone cannot guarantee the accuracy or integrity of the data. Without authentication, there is no way to verify that the results of a decryption are the same as original, unencrypted data. When combined with a third party, authentication provides a means of committing to a version of file, with no way to undetectably modify a file *a posteriori*. Section 302 of SOX results in CFOs and CEOs having a vested interest in the integrity of their financial records, as they must "certify" that their financial reports fairly represent the condition of their company. Authentication provides a way to bind an individual to their data, making the repudiation of data impossible. Lastly, authentication allows a company to prove to a customer that they are who they say they are, meeting the pretexting requirements in the GLBA.

1.8 Contributions

In this dissertation we make three contributions to the field of regulatory compliance storage technologies. The first is ext3cow, a platform for compliance with the versioning, audtitability and real-time disclosure requirements of electronic record retention legislation. Ext3cow is a file system that provides a unique time-shifting interface, permitting a real-time and continuous view of data in the past. For our second contribution, we add secure deletion to ext3cow, a method of permanently destroying data stored on magnetic media used to protect user privacy and limit a company's liability. Our solution is unique to versioning file systems, providing finer grained deletion and orders of magnitude better performance over existing techniques. Further, our secure deletion algorithms provide authenticated encryption, a transform that keeps data both private *and* authentic. Lastly, we introduce constructs that create, manage, and verify digital audit trails for versioning file systems. Using our model, auditors may efficiently verify the contents of a file system, meeting the authenticity requirements of electronic record legislation, such as Sarbanes-Oxley and Gramm-Leach-Bliley. By using I/O efficient parallel message authentication codes, sequences of versions may be easily authenticated and bound to a file system hierarchy, providing a complete authentic version history of a file system.

Chapter 2

Ext3cow: A Time-Shifting File System for Regulatory Compliance

> "Time is an illusion. Lunchtime doubly so."
>
> *–Douglas Adams*

Recent legislation makes new requirements of storage systems. The ext3cow file system, built on the popular ext3 file system, provides an open-source file versioning and snapshot platform for compliance with the versioning and audtitability requirements of electronic record retention legislation. Ext3cow provides a *time-shifting* interface that permits a real-time and continuous view of data in the past. Time-shifting does not pollute the file system namespace nor require snapshots to be mounted as a separate file system. Further, ext3cow is implemented entirely in the file system space and, therefore, does not modify kernel interfaces or change the operation of other file systems. Ext3cow takes advantage of the fine-grained control of on-disk and in-memory data available only to a file system, resulting in minimal degradation of performance and functionality. Experimental results confirm this hypothesis; ext3cow performs comparably to ext3 on many benchmarks and on trace-driven experiments.

2.1 Introduction

To address the versioning and auditability needs of regulated storage, we have developed *ext3cow*, an open-source disk file system based on the third extended file system (ext3). Ext3 [16] is the Linux default file system based on the Minix file system [114] and influenced by the Fast File System (FFS) [73]. Ext3 has become robust and reliable, providing reasonable performance and scalability for many users and workloads [15]. 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 the *time-shifting* interface, which 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. Other file systems that provide fine-grained access to versions do so by modifying kernel interfaces [26, 80, 101]. 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 [52] or via mounting separate logical volumes [111, 112]. Some disk file systems provide no interface to versions, restricting versioning to internal use only [98, 105].

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`. Furthermore, snapshots fit well into an information lifecycle management (ILM) framework. ILM is a policy-based scheme for managing the lifetime of electronic information, including time-sensitive data migration and consolidation, backups and restoration, disaster recovery, and long-term archiving. 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) [61]. 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 [75, 99]). 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 systems 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 [48]. Experience with HIPAA [60] 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.

In the remainder of this chapter, we present the details of the time-shifting interface and describe how file system data structures were retrofitted to support ext3cow's feature set in a disk file system. We present benchmarks and trace-driven experiments that show that versioning has a minor effect on the file system performance. On most micro-benchmarks, ext3cow meets the performance of ext3.

2.2 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 [19, 42, 49, 53] and for archiving and data mining [91]. File versioning, creating new logical versions on every disk write or on every open/close session, is used for tamper-resistant storage [111, 112] and file-oriented recovery from deletion [26, 80, 101]. Both techniques speed recovery and limit exposure to data losses during file system failure [52, 105]. A range of snapshot implementations exist, both at the logical file system level [42, 53, 54, 91, 101] and the disk storage level [37, 51, 54, 112].

File system versioning and snapshot have been used to recover from failure. FFS [70, 72, 73] takes snapshots to create a quiescent file system on which to perform on-line file system integrity checking. Writes that occur during a check are logged to a special snapshot file to which file system blocks are copy-on-written. FFS does not provide an interface to access file snapshots on-line. WAFL [52] also uses snapshots for recovery. In addition, 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 [98, 105] and Spiralog [43, 58] 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 [53, 78], the Episode file system [19], Plan-9 [89, 90], and SnapMirror [85] use snapshot with copy-on-write techniques as a method to perform quick, low-bandwidth backups in an on-line fashion. Venti [91] 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.* [17] and Azagury *et al.* [3].

Cedar [42, 49, 104] is the first example of a file system that maintains versions of a file over time. Versions are shared among file system users. Each write operation creates a new version that has a unique name to the file system, *e.g.* /home/user/ext3cow.tex!3 represents the third version. Each version of a file is autonomous, with no shared data between versions; sharing a file requires transferring all blocks of a file. Similar approaches were used by VMS [27, 69] and TOPS [79].

The Elephant file system [101] is the first file system to include a variety of user-specified retention policies similar to user-space version control tools such as RCS [95, 115], PRCS [66] and CVS [38, 44]. Elephant attempts to make intelligent decisions about which versions to keep, an approach taken by some on-line configuration management tools like CPCMS [107]. Elephant provides an intuitive, date-oriented interface. It is implemented as a replacement for the BSD VFS layer and provides versioning to all on-disk file systems that support its interface. Wayback [26] uses a similar versioning paradigm.

In the Comprehensive Versioning File System (CVFS) [111], all writes to the server, in a client/server storage system, are versioned, which provides an audit trail for security breaches. CVFS exists as a complete system, in which versions are accessed by mounting a point-in-time view of the file system over NFS [75, 99].

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

Table 2.1: Feature comparison of versioning file systems.

To place our contributions in context with respect to recent versioning file system research, Table 2.1 compares the features of ext3cow to CVFS [111], Elephant [101], Wayback [26], WAFL [52], and log-structured file systems (LFS) [98, 105]. We restrict this treatment to file systems, omitting versioning archives [4, 91], because we are concerned with interactive versioning in the regulatory environment. We also omit VersionFS [80] 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. Table 2.1 also indicates that log-structured file systems provide an attractive alternative; ext3cow's features could be achieved by adding the time-shifting interface to an LFS. However, one of our principal goals in building ext3cow for the regulatory environment is secure deletion (Chapter 3), which obviates the use of the log-structured layout. The write policy of LFS spreads data from a single file throughout the log. The efficiency of our secure deletion architecture (Chapter 3 [87, 88] relies on the file system clustering data and metadata (indirect blocks) so that a small amount of secure overwriting [45] securely deletes a large amount of data.

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

```
[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 2.1: Creating snapshots and accessing data in the past in ext3cow.

2.3 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.

Previous interfaces fail to fulfill our requirements for versioning in the regulatory environment. Some require old data to be accessed through a separate mount point [37, 111, 112], which prevents browsing in the existing file namespace to locate objects and then shifting those objects into the past. Others use arbitrary version numbers to access old data [27, 42, 49, 69, 79], *e.g.* access the file four versions back. These interfaces make sense for daily snapshots, but do not generalize to file versioning or more frequent snapshots. WAFL [52] uses namespace tunnels from the present to the past, that accesses the snapshot version of the current directory. While this permits browsing for files in the present and then shifting those files to the past, it does not handle multiple versions gracefully.

We describe the operation of the time shifting interface through the example of Figure 2.1. 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, 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 have developed shell extensions in a *Time-Traveling File Manager* (TTFM), which supports a variety of time and naming formats to help users browse versions (Section 2.8).

The time-shifting interface meets our 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.

```
[user@machine] snapshot /usr/bin
Snapshot on /usr/bin 1057866367
[user@machine] ln -s /usr/bin@1057866367 /usr/bin.preinstall
[user@machine] /usr/bin.preinstall/gcc
```

Figure 2.2: Creating distinguished (named) snapshots in ext3cow.

As an interface, time shifting is useful but not complete. We do not wish to require users to remember when they created versions. Thus, we allow users to tag or enumerate all versions of a file, reporting versions and their scope (creation and replacement time).

Distinguished snapshots may be created using links, which allows time-shifting to emulate the behavior of systems that put snapshots in their own namespaces or mount snapshot namespaces in separate volumes. For example, an administrator might create a read-only version of a file system prior to installing new software (Figure 2.2). If installing software breaks `gcc`, the administrator can use the old `gcc` through the mounted snapshot. Because `@` is a legal file system symbol, the link can be placed anywhere in the namespace, even within another file system. Hard links may also be used to connect a name directly to an old inode. This example illustrates that time-shifting is inherited from the parent directory, *i.e.* the entire subtree below `/usr/bin.preinstall` is scoped to the snapshot.

The time-shifting interface imposes some restrictions. Currently, the use of seconds since the Epoch limits (named) snapshots to one per second. For systems that use snapshot as part of recovery [52], sub-second granularity may be necessary. Because ext3cow, like ext3, uses a journal for file system recovery, we found no emergent need for sub-second snapshot. Furthermore, upcoming support for microsecond granularity time in ext3 will remove all limitations on the frequency of snapshots in ext3cow.

2.4 Metadata Design

The metadata design of ext3cow supports the continuous time notion of the time-shifting interface within the framework of the data structures of the Linux VFS. Unlike many other snapshot file systems [26, 80, 100, 101, 123], ext3cow does not interfere or replace the Linux common file model, therefore, it integrates easily, requiring no changes to the VFS data structures or interfaces. Modifications are limited to on-disk metadata and the in-memory file system specific fields of the VFS metadata. Ext3cow adds metadata to inodes, directory entries, and the superblock that allows it to scope requests from any point-in-time into a specific file version and support scoping inheritance.

2.4.1 Superblock

Implementing snapshot requires some method of keeping track of the *snapshot epoch* of every file in the system. We place a system-wide *epoch counter*, stored in the on-disk and in-memory superblock, as a reference for marking versions of a file. The counter is a 32-bit unsigned integer, representing the number of seconds passed since the Epoch. Using one second granularity, the 32-bit counter allows us to represent approximately 132 years of snapshots. We choose the same representation of time as does ext3. When ext3 adopts microsecond granularity times, ext3cow will be able to represent arbitrarily fine-grained epochs in the superblock.

To capture a point-in-time image of a file system the superblock epoch number is updated atomically to the current system time. Creating new file versions is not done at the time of the snapshot, but, rather, at the next operation that modifies the data or metadata of an inode, *e.g.* a write, truncate, or attribute change. Snapshots may be triggered internally by the file system or by an `ioctl` call made through the user-space snapshot utility.

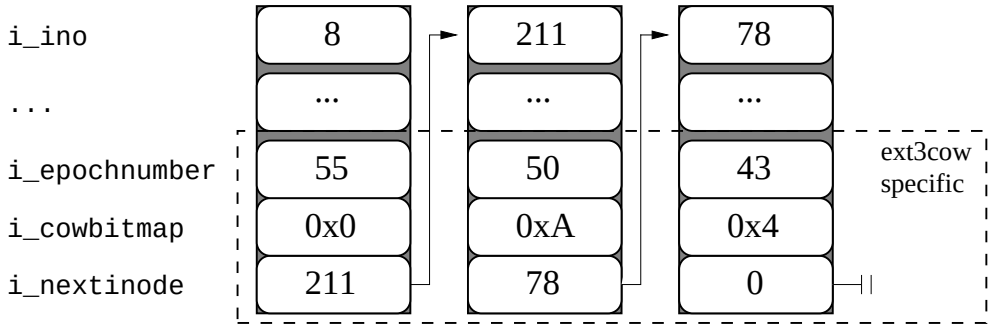


Figure 2.3: Both on-disk and in-memory inodes were retrofitted to support snapshot and copy-on-write by adding three fields: an inode epoch number, a copy-on-write bitmap, and field pointing to the next inode in the version chain.

2.4.2 Inodes

Inode versions identify how a file's data and attributes have changed between snapshots. For each system-wide snapshot, a file may have an inode that describes it in that epoch. A file that has not changed during an epoch shares an inode with the previous epoch(s). While space is very tight in the 128 byte ext3 on-disk inode, we were able to squeeze in an additional 20 bytes of information by removing empty fields used for disk sector alignment, as well as fields for the HURD operating system, which is not currently supported. Future versions of ext3 will expand the inode size to 256 or 512 bytes, eliminating all practical space constraints [116].

Three fields were added to both the on-disk and in-memory representation of the inode (Figure 2.3). A 32-bit `i_epochcounter` describes to which epoch an inode belongs. When writing data to an inode, the system updates the `i_epochcounter` to the system epoch counter. The `i_cowbitmap` maintains the block-versioning state of a file and is described in detail in Section 2.6.2. Lastly, we have added a pointer to the next version of an inode with the `i_nextinode` field.

Ext3cow supports both system-wide snapshots and individual file versions, by allowing a snapshot to be taken on a per-file basis. A variant of the snapshot utility, which takes a name as

an argument, sets a file epoch to the current time. An individually versioned inode has the property that its `i_epochcounter` exceeds the system-wide epoch. On write, ext3cow detects this condition and performs copy-on-write based on the file's epoch rather than the system epoch. A subsequent snapshot ends this condition and creates another new copy-on-write version of the file.

2.4.3 Directory Entries and Naming

Directories in ext3 and ext3cow are implemented as inodes in which the data blocks contain directory entries. Ext3cow versions directory inodes in the same manner as file inodes. The directory entries are versioned by adding scoping metadata to the directory entry (*dirent*). In ext3, a directory entry contains an inode number, a record length, a name length, and a name. To this, we add a *birth epoch* and a *death epoch* that determine the times during which a name is valid. Extending the directory entry is trivial and under no space constraints, because its length already varies in order to handle names and name deletions. Because directory entries are scoped to an epoch range, names that have been unlinked, and, therefore, given a death epoch, may be reused in a future context to represent a new file. Ext3cow only deletes one class of files; it permanently removes files that are unlinked in the same epoch in which they were created.

Retaining file names in ext3cow does not increase the directory size when compared with ext3. Both systems unlink names by increasing the record length of the preceding directory entry to span the deleted entry, an approach taken by similar file systems such as FFS [71]. In ext3cow, the space for unlinked names are not reused, nor are directories compacted. In contrast, FFS reuses space only after all names in a fixed sized chunk are unlinked. Neither approach is particularly attractive. Like both ext3 and FFS, ext3cow will benefit from efficient directory indexing data structures, which is a planned improvement [116].

2.5 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 (Section 2.3). 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.

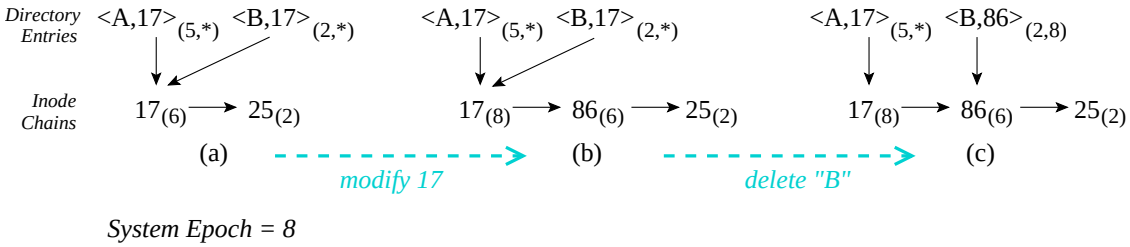


Figure 2.4: An example of names scoping to inodes over time.

2.5.1 Scoping Inodes

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 2.4(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 2.4(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.

2.5.2 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 2.4 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 2.4(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.

One concern with our scoping data structures is the linear growth of the inode chains over time. For frequently written files, each snapshot represents a new link in the chain and, thus, accesses to versions in the distant past may be prohibitively expensive. While file systems have a history of linear search structures, *e.g.* directories in ext3, we find the situation unacceptable and amend it.

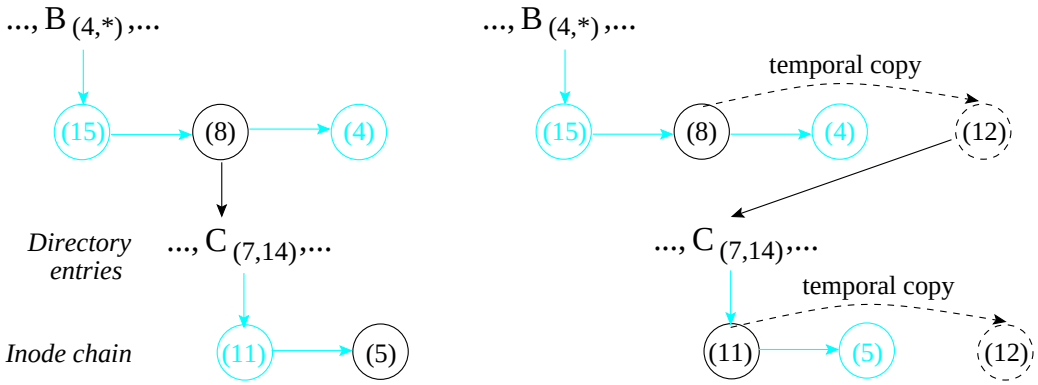
Ext3cow restricts version chains to a constant length through birth/death directory entry scoping. When the length of a version chain meets a threshold value, ext3cow terminates that chain by setting the death epoch of the directory entry used to access this chain to the current system epoch and creates a new chain (of length one) by creating a duplicate directory entry with a birth epoch equal to the system epoch. The stability of inodes ensures that other directory entries linking to the same data find the new chain. Data blocks may be shared between inodes in the two chains. A long-lived, frequently-updated file is described by many short chains rather than a single long

chain. While directory entries are also linear-search structures, this scheme increases search by a constant factor. It will improve the performance of version search from $O(n)$ to $O(1)$ when ext3 adopts extensible hashing for directories.

2.5.3 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 2.5(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 2.5(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.



(a) Incorrect scoping.

(b) Correct scoping with temporal vnodes.

Figure 2.5: 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.

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.

2.6 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` [71]) 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

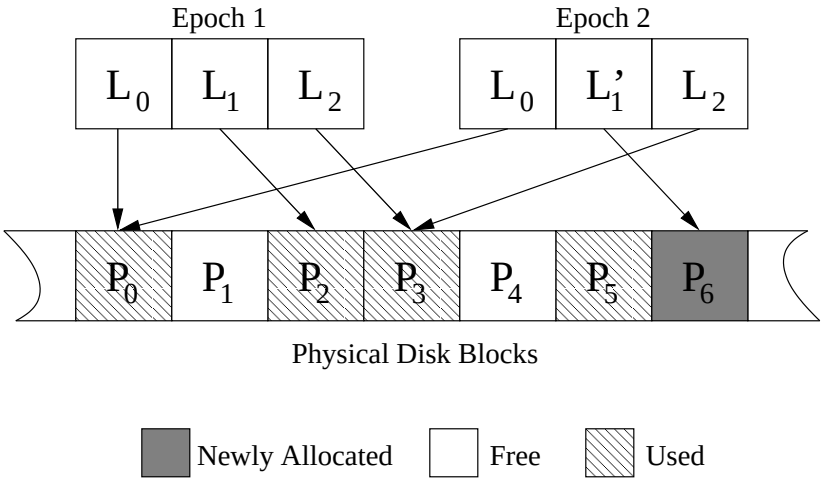


Figure 2.6: An example of copy-on-write. The version from epoch 2 updates logical block L_1 into L_1'. Ext3cow allocates a new physical (disk) block P_6 to record the difference. All other blocks are shared.

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 2.6).

When the most recent version of a file precedes the system epoch in time, any change to that file creates a new physical version. The first step is to duplicate the inode, as discussed in Section 2.5. 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.

2.6.1 Memory Management

We isolate the copy-on-write function to the on-disk file system; we do not trespass into kernel components such as the VFS or page cache. To achieve this isolation, ext3cow leverages Linux's multiple interfaces into memory; memory pages that hold file data are comprised of file system buffers, which map in-memory blocks to disk blocks. Ext3cow performs copy-on-write by re-allocating the file-system blocks that represent the storage for a buffer. In Linux, the VFS passes a `write` system call to the on-disk file system prior to updating a page in memory. This allows a file system to map the file offset to a memory address and bring data into the cache from disk as needed. In ext3cow, we take this opportunity to determine if a file block needs to be copy-on-written and to allocate a new backing block when necessary. Ext3cow replaces the disk block that backs (provides storage for) an existing block in the buffer and marks the buffer dirty. Then, the `write` call proceeds using the same memory page. At some point in the future, the buffer manager writes the dirtied blocks to disk as part of cache management. The actual copy is created at this time. Through re-allocation, ext3cow creates on-disk copies of blocks without copying data in memory.

The copy-on-write design preserves system cache performance and minimizes the I/O overheads associated with managing multiple versions. Ext3cow consumes no additional memory and does not pollute the page cache with additional data. Additionally, for data blocks, copy-on-write incurs no additional I/O, because the dirtied buffers are updated by the `write` call and need to be written back to disk anyway. The only deleterious effect of copy-on-write is I/O for indirect blocks, which do not necessarily get updated as part of a `write` in ext3.

2.6.2 Copy-on-write State bitmaps

Ext3cow embeds bitmaps in its inodes and indirect blocks that allow the system to record which blocks have had a copy-on-write performed. In the inode, ext3cow uses one bit for each direct block, one for the indirect, doubly-indirect, and triply-indirect block respectively. A bit of value 0 indicates that a new block needs to be allocated on the next write and bit value 1 indicates that a new allocation of this block has been performed within the current epoch and that data may be updated in place. Ext3cow zeroes the entire bitmap when duplicating an inode. In an indirect block (resp. doubly or triply indirect block), the last eight 32-bit words of the block contain a bitmap with a bit for every block referenced in that indirect block, which are also zeroed when creating a copy-on-write version of the indirect block. The bitmap design allows the bitmaps to be updated lazily – only when data are written, not on snapshot.

Because bitmaps borrow space in indirect blocks, the design reduces the maximum file size. However, the loss is less than 10%. Ext3cow represents files up to 15,314,756 blocks in comparison to 16,843,020 blocks in ext3. While larger than 2^{32} bytes, Linux supports 64-bit file offsets. The upcoming adoption of quadruply indirect blocks [116] will remove practical file size limitations.

The bitmap design allows ext3cow to improve performance when truncating a file. Truncate is a frequent file system operation: applications often truncate a file to zero length as a first step when rewriting that file. On truncate, ext3 deallocates all blocks of a file. In contrast, ext3cow deallocates only those blocks that have been written in the current epoch. Other blocks remain allocated to be used in older versions of the file. Therefore, ext3cow skips deallocation for any blocks for which the corresponding state bitmap equals zero. For indirect blocks (resp. doubly or triply

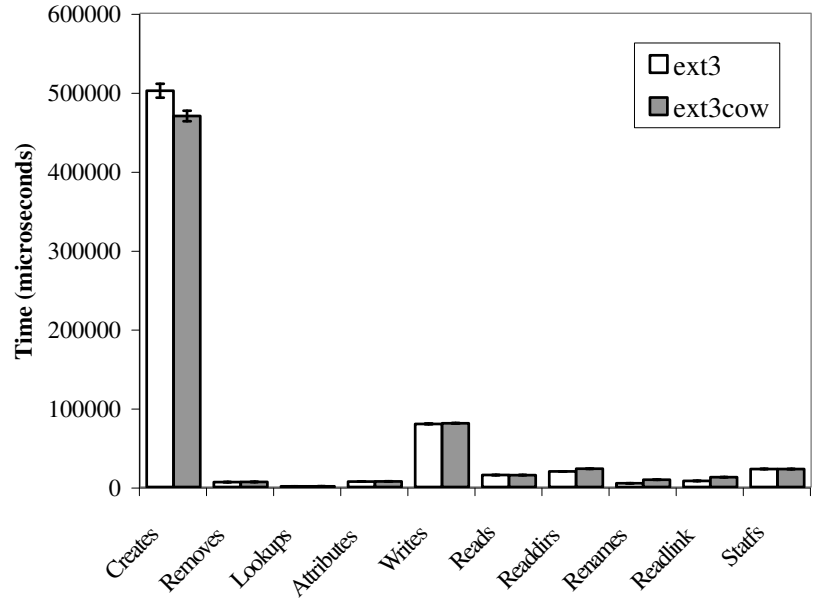
Operational Test	ext3	ext3cow
Test 1: Creates	501.90 ms	469.94 ms
Test 2: Removes	6.23 ms	6.49 ms
Test 3: Lookups	0.96 ms	0.96 ms
Test 4: Attributes	6.87 ms	7.19 ms
Test 5a: Writes	79.91 ms	80.65 ms
Test 5b: Reads	15.14 ms	15.10 ms
Test 6: Readdirs	19.72 ms	23.12 ms
Test 7: Renames	4.68 ms	9.22 ms
Test 8: Readlink	7.68 ms	12.46 ms
Test 9: Statfs	22.86 ms	22.76 ms

Table 2.2: Results from the "basic" tests of the Connectathon benchmark suite.

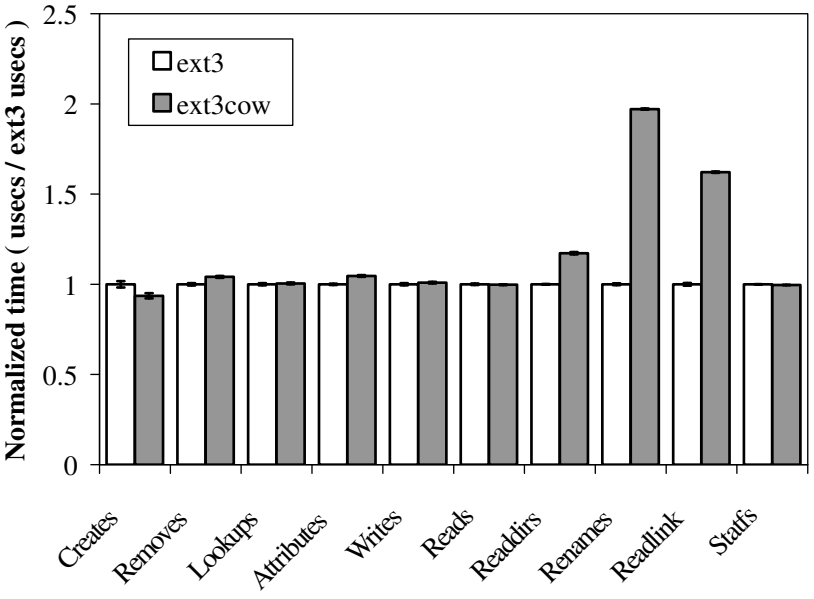
indirect blocks), ext3cow skips deallocation for the entire subtree underneath that block corresponding to the zero bit. In this way, ext3cow minimizes I/O to deallocate blocks and update free-space bitmaps during truncate.

2.7 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.



(a) Time by benchmark



(b) Time normalized to ext3

Figure 2.7: Results from the “basic” tests in the Connectathon benchmark suite. All data are shown with 95% confidence intervals.

2.7.1 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. Table 2.2 shows the average cumulative time to perform each test. We present the same data as bar graphs in both absolute time values (Figure 2.7(a)) and time normalized to the performance of ext3 (Figure 2.7(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. The Attributes and Removes tests navigate a name tree, operating on files rather than inodes directly. The small difference in performance comes from overhead on naming operations. On create, names do not need to be parsed for time scoping.

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. Ext3cow needs to

check the copy-on-write bitmap to determine whether a block should be copied the first time a block is written. Subsequent writes to that (dirty) block do not need to check again. The benchmark truncates and rewrites the file anew on each trial, and, therefore, does not exercise this feature.

String operations to support versioning result in ext3cow under-performing ext3 on tests dominated by name operations. During lookup, ext3cow parses every name looking for the `@` version specifier. Ext3cow takes the string prior to `@` as a file name and uses the remainder of the string for scoping. 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.

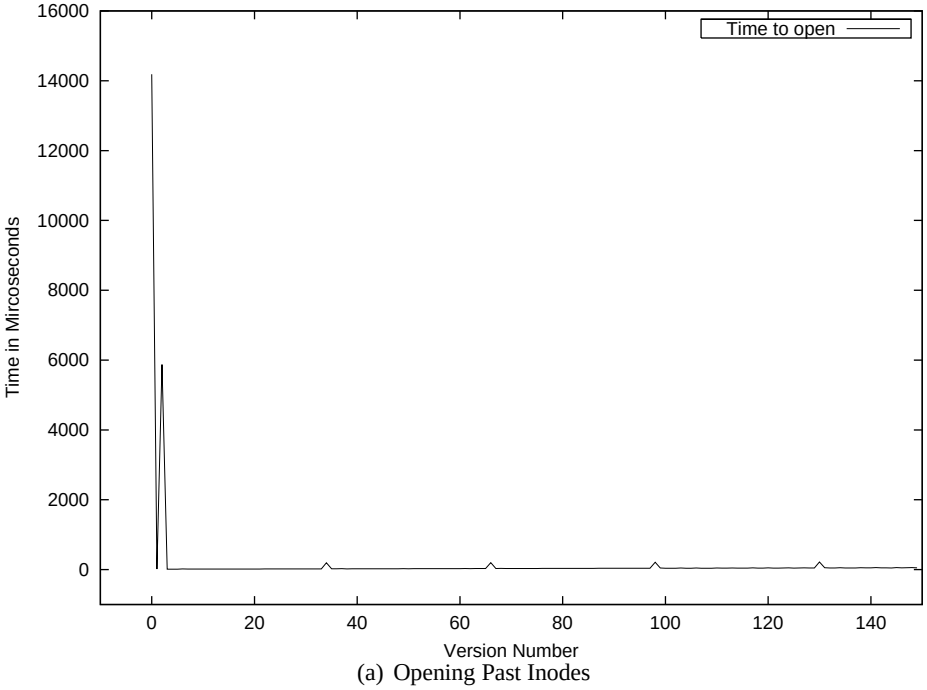
Test 6 (Readdirs) shows the overhead of scoping names in directories. The system does not parse strings or interpret the `@` symbol during this test. Directory names are read and returned to the calling function without interpretation. The overhead comes from directory entry scoping only. Ext3cow examines the birth and death epoch of every record that it reads. This overhead is modest in the benchmark, but might be larger in practice when ext3cow needs to consider more names in a directory – those from previous epochs as well as current epoch. In total, micro-benchmark results indicate that ext3cow performs comparably to ext3 on data and inode operations and slightly worse on name operations.

Performance in the Past

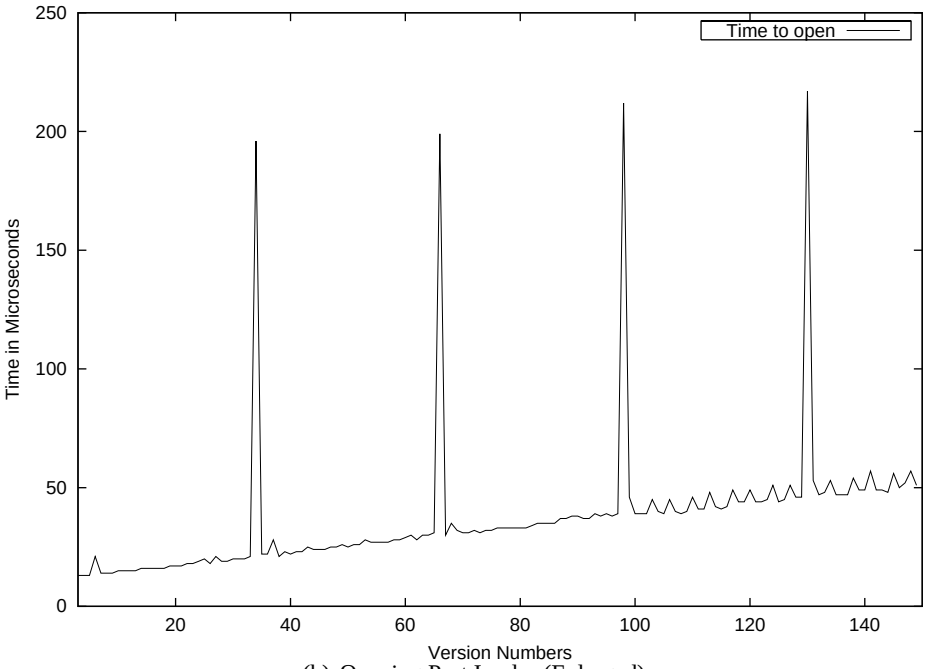
To capture the effect of multiple versions on performance, we modified the Connectathon benchmark to measure the time to open a series of 150 versions of a file from youngest to oldest. These versioned inodes were created consecutively and, therefore, ext3cow lays them out near-contiguously in block groups. Figure 2.8(a) shows the results of this test on a cold cache. To access the first inode, the system incurs two disk seek penalties for I/O: one to lookup the inode by name and one to access the inode. Almost all subsequent inode accesses are served out of different caches. Figure 2.8(b) shows a closeup of Figure 2.8(a) with large values filtered out. The baseline represents fetches out of the file-system cache, with linear scaling because accessing the k^{th} version traverses k inodes in cache. Every 32 inodes, the file system fetches a new group (4KB) of inodes from disk, which, based on the low-latency, seem to be served out of the disk's cache. Having fetched the next group of inodes, subsequent accesses to these inodes are served in the file system cache.

We attempted many "worst-case" versions of this experiment by artificially aging a file system. In one experiment, we create a block group worth of inodes ($>$16,000) between successive inodes in a chain. The goal was to subvert and render ineffective ext3's inode clustering. The results of all experiments were indistinguishable from the original experiment, showing an initial penalty for I/O and subsequent access out of caches. Our many attempts to "game" ext3cow were rendered ineffective by the combination of placement policies, read-ahead, and disk (track) caching.

We also conducted experiments that flush the cache between accesses to the k^{th} and $k+1^{th}$ versions. In this experiment, each inode in the chain takes approximately 14 ms to access, because I/O dominates in-memory operations. Performance grows linearly in the number of versions.



(a) Opening Past Inodes



(b) Opening Past Inodes (Enlarged)

Figure 2.8: The time to open 150 versions of a file.

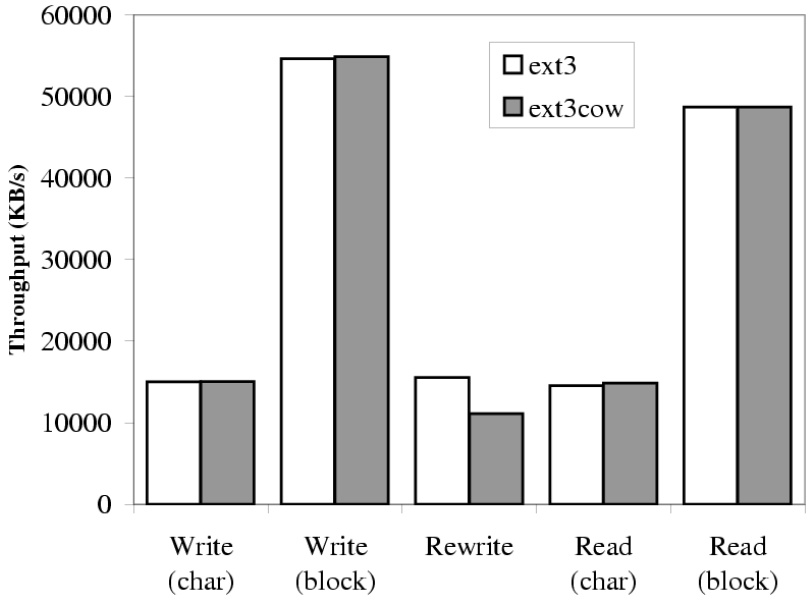


Figure 2.9: Results from the Bonnie++ file system benchmark.

2.7.2 Bonnie++

Bonnie++ is a popular performance benchmark that quantifies five aspects of file system performance based on observed I/O bottlenecks in a Unix-based file system. Bonnie performs a series of test on files of a known size. In our experiments, two, one gigabyte files. This insures I/O requests are not served out of a disk cache, requiring disk access. For each test, the benchmark reports throughput, measured in kilobytes processed per second. The first test measures the rate of sequential character output, while the second test measures sequential block output. The files, in the third test, are then sequentially read and rewritten. Lastly, tests four and five measure the sequential input, by character and by block.

The results of an ext3 and ext3cow Bonnie++ comparison are presented in Figure 2.9. Ext3cow performs comparably with ext3 in all but the rewrite experiment. This slight degradation is due to the copy-on-write bitmap operations that must be performed when rewriting a buffer.

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 2.3: 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.

2.7.3 Trace-driven Experiments

To examine the effect of snapshots on metadata allocation, we used four months of Berkeley system call traces [97] to populate a file system and performed an off-line analysis, identifying the type and amount of allocation. By aging a file system, we more accurately measure and analyze real-world performance [109]. 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. The traces contain no file names or directory hierarchy, but do record file creations and directory traversals. For our experiments, we use these operations to infer a directory hierarchy and create file names. Our experiment maintains a consistent mapping of the files we create to the file and device identifiers in the trace. This ensures that operations to the same file in the trace are performed to the same file in our experiment.

Table 2.3 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 [111]. Of the 10.2% increase in metadata, 4.7% comes from versioned directory inodes.

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. We expect the overhead rate of 27% to be stable over time. The majority of this overhead comes from recent files (older than 1 minute and younger than 1 hour) and, thus, updates to old data do not make up a large portion. Similarly, updates from files older than a day make up only 3.8% of the total 27%. Inode overheads are smaller than block overheads. Directory inode overheads are much greater, ranging to 421%. 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.

2.8 Availability

Ext3cow is stable and ready for use under the GNU Public License. It is available for download at *http://www.ext3cow.com*. The site, to date, has received thousands of visitors and hundreds of downloads. Beyond its use as an effective file system for end users, ext3cow is being employed as the platform for additional systems research. The Zap project [83] is using ext3cow as the foundation for a virtualization layer that provides groups of processes a consistent view of a system. Ext3cow is also the basis of on-going research on the aging of versioning file systems at U.C. Santa Cruz and practical time-shifting features at U.C. Berkeley. A number of free-lance programmers have asked for support in building network storage devices based on ext3cow. The Time Traveling File Manager (TTFM) [92] is a graphical user interface created for ext3cow that provides

an intuitive interface for easily navigating a time-shifting file system. TTFM's features include a date slider, allowing an easy navigation into views of the past, a tool for version comparison, and the ability to launch a shell in a past file system context.

Chapter 3

Secure Deletion for a Federally Compliant Storage System

> "Three may keep a secret, if two of them are dead."
>
> *– Benjamin Franklin*

Under Section 802 of the Sarbanes-Oxley Act, provisions of the HIPAA Privacy Rule, and the GLBA Safeguard Rule, data should have limited lifetime to limit the liability of a company and to protect the privacy of clients and patients. In this chapter, we present algorithms and an architecture for the secure deletion of individual versions of a file. The principal application of this technology are versioning file systems used for federally compliant storage; it is designed to eliminate data after a mandatory retention period and to protect the privacy of financial or medical records. However, it applies to any storage system that shares data between files. We compare two methods for secure deletion that use a combination of authenticated encryption and secure overwriting. We also discuss implementation issues, such as the demands that secure deletion places on version creation and the composition of file system metadata. Our results show that new secure deletion techniques perform orders of magnitude better than previous methods.

3.1 Introduction

Versioning storage systems are increasingly important in research and commercial applications. 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 a file, while preserving all other data in the system.

Secure deletion is valuable to security conscious users and organizations. It protects the privacy of user data and prevents the discovery of information on retired or sold computers. Traditional data deletion, or "emptying the trash", simply frees blocks for allocation at a later time; the data persists, fully readable and intact. Even when data are overwritten, information may be reconstructed using expensive forensic techniques, such as magnetic force microscopy [125].

We are particularly interested in using secure deletion to limit liability in the regulatory environment. By securely deleting data after they have fallen out of regulatory scope, *e.g.* seven years for corporate 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 storage systems that share data among files, such as versioning file systems [26,52,80,86,101,111] and content-indexing systems [4, 81, 91].

The preferred and accepted methods for secure deletion in non-data sharing systems include: repeatedly overwriting data, such that the original data may not be recovered [45]; and, encrypting a file with a key and securely disposing of the key to make the data unrecoverable [13].

Block sharing hinders key management in encrypting systems that use key disposal. If a system were to use an encryption key per version, the key could not be discarded, as it is needed to decrypt shared blocks in future versions that share the encrypted data. To realize fine-grained secure deletion by key disposal, a system must keep a key for every shared block, resulting in an onerous number of unmanageable keys. Fewer keys allow for more flexible security policies [59].

Secure overwriting also has performance concerns in versioning systems. In order to limit storage overhead, versioning systems often share blocks of data between file versions. 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.

Our contributions include two methods for the secure deletion of individual versions that minimize 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* of 128 bits. We collect and store stubs contiguously in a file system block so that overwriting a 4K block of stubs deletes the corresponding 1MB of file data, even when file data are non-contiguous. Unlike encryption keys, stubs are not secret and may be stored on disk. Our methods do not complicate key management. We also present a method for securely deleting data out-of-band, a construct that lends itself to multiple parties with a shared interest in a single piece of data and to off-site back-ups.

To our knowledge, ext3cow is the first file system to adopt authenticated encryption (AE), which provides both privacy and authenticity. Authenticity is essential to ensure that the data have not changed between being written to disk and read back. Particularly in environments where storage is virtualized or distributed and, thus, difficult to physically secure. Authenticated encryption requires message expansion – ciphertext are larger than the plaintext – which is an obstacle to its adoption. Encrypting file systems have traditionally used block ciphers, which preserve message size, to meet the alignment and capacity constraints of disk drives [10, 59, 122]. In practice, additional storage must be found for the expanded bits of the message. Our architecture creates a parallel structure to the inode block map for the storage of expanded bits of the ciphertext and leverages this structure to achieve secure deletion. Message expansion is fundamental to our deletion model.

Secure deletion and authenticated encryption has been implemented in the ext3cow versioning file system, designed for version management in the regulatory environment [86]. Experimental results show that our methods for secure deletion improve deletion performance by several orders of magnitude. Also, they show that metadata maintenance and cryptography degrade file system performance minimally.

3.2 Related Work

Secure Deletion

Garfinkel and Shelat [41] survey 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 lawsuits 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 [45] gives a technique that takes 35 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 [41]). This technique has been implemented in user-space tools and in a Linux file system [5]. Secure overwriting has also been applied in the semantically-smart disk system [108].

For file systems that encrypt data on disk, data may be securely deleted by "forgetting" the corresponding encryption key [13]; 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. The actual disposal of the encryption key may involve secure overwriting.

There are many user-space tools for secure deletion, such as `wipe`, `eraser`, and `bootandnuke`. These tools provide some protection when securely deleting data. However, they may leak information because they are unable to delete metadata. They may also leak data when the system truncates files. Further, they are difficult to use synchronously because they cannot be interposed between file operations.

The importance of deleting data has been addressed in other system components. A concept related to stub deletion has been used in memory systems [30], which erase a large segment of memory by destroying a small non-volatile segment. Securely deallocating memory limits the exposure of sensitive data [18]. Similar problems have been addressed by Gutmann [46, 47] and Viega [117].

Secure Systems

CFS [10] was an early effort that added encryption to a file system. In this user-space tool, local and remote (via NFS) encrypted directories are accessed via a separate mount point. All file data and metadata in that directory are encrypted using a pre-defined user key and encryption algorithm. CFS does not provide authenticated encryption.

NCryptfs [122] is a cryptographic file system implemented as a stackable layer in FiST [124]. The system is designed to be customizable and flexible for its users by providing many options for encryption algorithms and key requirements. It does not provide authenticated encryption.

Cryptoloop uses the Linux cryptographic API [77] and the loopback interface to provide encryption for blocks as they are passed through to the disk. While easy to administer for a single-user machine, cryptographic loopback devices do not scale well to multi-user systems.

Our implementation of encryption follows the design of the CryptoGraphic Disk Driver (CGD) [36]. CGD replaces the native disk device driver with one that encrypts blocks as they are transfered to disk.

The encryption and storage of keys in the random-key encryption scheme resembles lock-boxes in the Plutus file system [59] in which individual file keys are stored in lock-boxes and sealed with a user's key.

Cryptography

Secure deletion builds upon cryptographic constructs that we adapt to meet the demands of a versioning file system. The principal methods that we employ are the all-or-nothing transform [94], secret-sharing [106], and authenticated encryption [8]. Descriptions of their operation and application appear in the appropriate technical sections.

3.3 Secure Deletion with Versions

We have a developed an approach to secure deletion for versioning systems that minimizes the amount of secure overwriting, eliminates the need for data block contiguity, and does not increase the complexity of key management.

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

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

that takes a data block (B_i), a key (k) and a nonce (N) and creates an output that can be partitioned into an encrypted 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. When the key (k) remains private, the transform acts as an authenticated encryption algorithm. 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)*, which models the situation in which a key is exposed, *e.g.* by subpoena. The stub reveals nothing about the key or the data, and, thus, stubs may be stored on the same disk. It may be possible to recover securely deleted data after the key has been exposed by a brute-force search for the stub. However, this is no easier than a brute-force search for a secret key and is considered intractable.

A distinct advantage of our file system architecture is the use of authenticated encryption [8]. Authenticated encryption is a transform by which data are kept both private *and* authentic. Many popular encryption algorithms, such as AES, by themselves, provide only privacy; they cannot guarantee that the decrypted plaintext is the same as the original plaintext. When decrypting, an authenticated encryption scheme will take a ciphertext and return either the plaintext or an indication the ciphertext is invalid or unauthentic. A common technique for authenticated encryption is to

combine a message authentication code (MAC) with a standard block cipher [8]. However, single pass methods exist [96].

Authenticated encryption is a feature not provided by encrypting file systems to date. This is because authenticated encryption algorithms expand data when encrypting; the resulting cipherblock is larger than the original plaintext. This causes a mismatch in the block and page size. File systems present a page of plaintext to the memory system, which fills completely a number of sectors on the underlying disk. The AE encrypted ciphertext is larger than and does not align with the underlying sectors. (Other solutions based on a file system or disk redesign are possible). Expansion results in a loss of transparency for the encryption system. We address the problem of data expansion and leverage the expansion to achieve secure deletion.

Our architecture for secure deletion with stubs does not complicate key management. It employs the same key-management framework used by disk-encrypting file systems based on block ciphers, such as Plutus [59] and NCryptfs [122]. It augments these to support authenticated encryption and secure deletion.

We present and compare two implementations of the keyed transform (f_k): one inspired by the all-or-nothing transform and the other based on randomized keys. Both algorithms allow for the efficient secure deletion of a single version. We also present extensions, based on secret-sharing, that allow for the out-of-band deletion of data by multiple parties.

3.3.1 AON Secure Deletion

The all-or-nothing (AON) transform is a cryptographic function that, given a partial output, reveals nothing about its input. No single message of a ciphertext can be decrypted in isolation without decrypting the entire ciphertext. The transform requires no additional keys. The original

Input: Data Block $d_1, \ldots, d_m$, 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) AON encryption

Input: Stub x_0, Ciphertext $x_1, \ldots, x_m$, Block ID id, Counter x, Encryption key K, MAC key M

1: $ctr_2 \leftarrow id||x||0||0^{128-|x|-|id|-1}$
2: $t \leftarrow x_0 \oplus \ldots \oplus x_m$
3: $c_1, \ldots, c_m \leftarrow \text{AES-CTR}_t^{ctr_2}(x_1, \ldots, x_m)$
4: $t' \leftarrow \text{HMAC-SHA-1}_M(c_1, \ldots, c_m)$
5: if $t' \neq t$ return $\perp$
6: $ctr_1 \leftarrow id||x||1||0^{128-|x|-|id|-1}$
7: $d_1, \ldots, d_m \leftarrow \text{AES-CTR}_K^{ctr_1}(c_1, \ldots, c_m)$

Output: Data Block $d_1, \ldots, d_m$

(b) AON decryption

Figure 3.1: Authenticated encryption and secure deletion for a single data block in a versioning file system using the all-or-nothing scheme.

intention, as proposed by Rivest [94], was to prevent brute-force key search attacks by requiring the attacker to decrypt an entire message for each key guess, multiplying the work by a factor of the number of blocks in the message. Boyko presented a formal definition for the AON transform [14] and showed that it meets the OAEP [8] scheme used in many Internet protocol standards. AON has been proposed to make efficient smart-card transactions [11, 12, 57], message authentication [35], and threshold-type cryptosystems using symmetric primitives [2].

The AON transform is the most natural construct for the secure deletion of versions. We aim to minimize the amount of secure overwriting. We also aim to not complicate key management. AON fulfills both requirements while conforming to our deletion model. The all-or-nothing property of the transform allows the system to overwrite any small subset of a data block to delete the entire

block; without all subsets, the block cannot be read. When combined with authenticated encryption, the AON transform creates a message expansion that is bound to the same all-or-nothing property. This expansion is the stub and can be securely overwritten to securely delete a block. Because the AON transform requires no additional keys, key management is no more complicated than a system that uses a block cipher.

We present our AON algorithm for secure deletion in Figure 3.1. The encryption algorithm (Figure 3.1(a)) takes as inputs: a single file system data block segmented into 128-bit plaintext messages ($d_1, \ldots, d_m$), a unique identifier for the block (id), a unique global counter (x), an encryption key (K) and a MAC key (M). To encrypt, the algorithm first generates a unique encryption counter (ctr_1) by concatenating the block identifier (id) with the global counter (x) and padding with zeros (Step 1). This counter is used as an initialization vector to the block cipher to prevent similar data blocks from encrypting to the same cipher block. The same counter and key combination should not be used more than once, so we use the block's physical disk address for id and the time in which it was written for x; both characteristics exist within an inode. An AES encryption of the data is performed in counter mode (AES-CTR) using a single file key (K) and the counter generated in Step 1 (ctr_1). This results in encrypted data ($c_1, \ldots, c_m$). The encrypted data are authenticated (Step 3) using SHA-1 and MAC key (M) as a keyed-hash for message authentication codes (HMAC). The authenticator (t) is then used as the key to re-encrypt the data (Step 5). It is this step that makes the authentication and encryption strongly non-separable. A second counter (ctr_2) is used to prevent repetitive encryption. A stub (x_0) is generated (Step 6) by XOR-ing all the ciphertext message blocks ($x_1, \ldots, x_m$) with the authenticator (t). The resulting stub is not secret, rather, it is an expansion of the encrypted data and is subject to the all-or-nothing property. The

ciphertext ($x_1, \ldots, x_m$) is written to disk as data, and the stub (x_0) is stored as metadata.

Decryption (Figure 3.1(b)) works similarly, but in reverse. The algorithm is given as inputs: the stub (x_0), the AON encrypted data block ($x_1, \ldots, x_m$), the same block ID (id) and counter (x) as in the encryption, and the same encryption (K) and MAC (M) keys used to encrypt. The unique counter (ctr_2) is reconstructed (Step 1), the authenticator (t) is reconstructed (Step 2) and then used in the first round of decrypting the data (Step 3). An HMAC is performed on the resulting ciphertext (Step 4) and the result (t') is compared with the reconstructed authenticator (t) (Step 5). If the authenticators do not match, the data are not the same as when they were written. Lastly, the data are decrypted (Step 7), resulting in the original plaintext.

Despite the virtues of providing authenticated encryption with low performance and storage overheads, AON encryption suffers from a guessed-plaintext attack. After an encryption key has been revealed, if an attacker can guess the exact contents of a block of data, she can verify that the data were once in the file system. This attack does not reveal encrypted data. Once the key is disclosed, 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 an 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 an individual had read and subsequently made attempts to destroy an incriminating email.

Input: Data Block $d_1, \ldots, d_m$, Block ID id, Counter x,
Encryption key K, MAC key M
2: $nonce \leftarrow id||x$
3: $c_1, \ldots, c_n \leftarrow \mathrm{AE}_k^{nonce}(d_1, \ldots, d_m)$
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 $c_1, \ldots, c_m$

(a) Random-key encryption

Input: Stub $c_0, t, c_{n+1}, \ldots, c_m$, Ciphertext $c_1, \ldots, c_n$,
Block ID id, Counter x, Encryption key K, MAC key M
1: $ctr \leftarrow id||x||0^{128-|x|-|id|}$
2: $t' \leftarrow \text{HMAC-SHA-1}_M(ctr, c_0, r)$
3: if $t' \neq t$ return $\perp$
4: $k \leftarrow \text{AES-CTR}_K^{ctr}(c_0)$
5: $nonce \leftarrow id||x$
6: $d_1, \ldots, d_n = \mathrm{AE}_k^{nonce}(c_1, \ldots, c_m)$
Output: Data Block $d_1, \ldots, d_n$

(b) Random-key decryption

Figure 3.2: Authenticated encryption and secure deletion for a single data block in a versioning file system using the random-key scheme.

3.3.2 Secure Deletion Based on Randomized Keys

As mentioned by Rivest [94], avoiding such a text-guessing attack requires that an AON transform employ randomization so that the encryption process is not repeatable given the same inputs. The subsequent security construct generates a random key on a per-block basis.

Random-key encryption is not an all-or-nothing transform. Instead, it is a refinement of the Boneh key disposal technique [13]. Each data block is encrypted using a randomly generated key. When this randomly generated key is encrypted with the file key, it acts as a stub. Like AON encryption, random-key encryption makes use of authenticated encryption, minimizes the amount of data needed to be securely overwritten, and does not require the management of additional keys.

We give an algorithm for random-key secure deletion in Figure 3.2. To encrypt (Figure 3.2(a)), the scheme generates a random key, k, (Step 1) that is used to authenticate and encrypt

a data block. Similar to the unique counters in the AON scheme, a unique nonce is generated (Step 2) to seed randomness when encrypting. Data is then encrypted and authenticated (Step 3), resulting in an expanded message. The algorithm is built upon any authenticated encryption (AE) scheme; AES and SHA-1 satisfy standard security definitions. To avoid the complexities of key distribution, we employ a single encryption (K) and MAC (M) key per file (the same keys as used in AON encryption) and use these keys to encrypt and authenticate the random key (k) (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) does not need to be securely overwritten to permanently destroy data.

An advantage of random-key encryption over AON encryption is its speed. For example, when the underlying AE is OCB [96], only one pass over the data is made and it is fully parallelizable. However, 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. We have developed another algorithm that requires no more bits than the underlying AE scheme. Unfortunately, this is based on OAEP and a Luby-Rackoff construction [64] and is only useful for demonstrating that space efficient constructions do exist. It is far too slow to be used in practice, requiring six expensive passes over the data.

3.3.3 Other Secure Deletion Models

Our secure deletion architecture was optimized for the most common deletion operation: deleting a single version. However, there are different models for removing data that may be more efficient in certain circumstances. These include efficiently removing a block or all blocks from an entire version chain and securely deleting data shared by multiple by parties.

Deleting a Version Chain

When a user wishes to delete an entire version chain, *i.e.* all blocks associated with all versions of a file, it may be more efficient to securely overwrite the blocks themselves rather than each version's stubs. This is because overwriting is slow and many blocks are shared between versions. For example, to delete a large log file to which data has only been appended, securely deleting all the blocks in the most recent version will delete all past versions.

AON encryption allows for the deletion of a block of data from an entire version chain. Due to the all-or-nothing properties of the transform, the secure overwriting of any 128 bits of a block results in the secure deletion of that block, even if the stub persists. Ext3cow provides a separate interface for securely deleting data blocks from all versions. If a deleted block was shared, it is no longer accessible to other versions, despite their possession of the stub.

Randomized-key encryption does not hold this advantage; 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. To remedy this, a key share (Section 3.3.3) could be stored alongside the encrypted data block. When the key share is securely overwritten, the encrypted data are no longer accessible in any version. However, this strategy is not practical in most file systems, owing to block size and alignment constraints. Storage for the key share must be provided and there is no space in the file system block. The shares could be stored elsewhere, as we have with deletion stubs, but need to be maintained on a per-file, rather than per-version, basis.

Secure Deletion with Secret-Sharing

The same data are often stored in more than one place. An obvious example of this are remote back-ups. It is desirable that when data fall out of regulatory scope, all copies of data are destroyed. Secret-sharing provides a solution.

Our random-key encryption scheme allows for the separation of the randomly-generated encryption key into n key shares. This is a form of an (n,n) secret-sharing scheme [106]. In secret-sharing, Shamir shows how to divide data into n shares, such that any k shares can reconstruct the data, but where $k-1$ shares reveals nothing about the data. We are able to compose a single randomly generate encryption key (k) from multiple key shares. An individual key share may then be given to a user with an interest in the data, distributing the means to delete data. If a single key share is independently deleted, the corresponding data are securely deleted and the remaining key shares are useless. Without all key shares, the randomly generated encryption key may not be reconstructed and decryption will fail.

Any number of randomly generated keys may be created in Step 1 (Figure 3.2(a)) and composed to create a single encryption key (k). To create two key shares, Step 1 is replaced with:

$$k \leftarrow \ell \oplus r$$

The stub (c_0) then becomes the encryption of any one key share, for example:

$$c_0 \leftarrow \text{AES-CTR}_K^{ctr}(\ell)$$

With an (n,n) key share scheme, any single share may be destroyed to securely delete the corre-

sponding data. The caveat being that all key shares must be present at the time of decryption. This benefits parties who have a shared interest in the same data. For example, a patient may hold a key share for their medical records on a smartcard, enabling them to control access to their records and also independently destroy their records without access to the storage system.

This feature extends to the management of securely deleting data from back-ups systems. Data stored at an off-site location may be deleted out-of-band by overwriting the appropriate key shares. In comparison, without secret-sharing, all copies of data would need to be collected and deleted to ensure eradication. Once data are copied out of the secure deletion environment, no assurance as to the destruction of the data may be made.

3.3.4 Security Properties

It is commonplace for systems based on "novel" security protocols to be broken. For example, the 802.11 WEP has been shown to be completely insecure after being deployed to millions of users [113]. *Ad hoc* security designs often fail careful analysis. A better approach is to build systems using proven security constructs and protocols. Proven constructs have been reduced to primitives that are believed to be secure because they have never been broken under intense scrutiny (such as AES) or new protocols that have been reduced to known secure protocols.

AON and random-key secure deletion were designed using only provably secure constructs and protocols and, therefore, are as secure as the underlying primitives. Provably secure is sometimes more expensive in performance and storage overhead, *e.g.* the size of the stubs. This is often the trade-off for proven security.

3.4 Architecture

We have implemented secure deletion in ext3cow [86], an open-source, block-versioning file system designed to meet the requirements of electronic record management legislation. Ext3cow supports file system snapshot, per-file versioning, and a time-shifting interface that provides real-time access to past versions. Versions of a file are implemented by chaining inodes together where each inode represents a version of a file.

3.4.1 Metadata for Secure Deletion

Metadata in ext3cow have been retrofitted to support versioning and secure deletion. For versioning, ext3cow employs a copy-on-write policy when writing data. Instead of overwriting old data with new data, ext3cow allocates a new disk block in which to write the new data. A new inode is created to record the modification and is chained to the previous inode. Each inode represents a single version and, as a chain, symbolizes the entire version history of a file. To support versioning, ext3cow "steals" address blocks from an inode's indirect blocks to embed bitmaps used to manage copy-on-written blocks. In a 4K indirect block (respectively, doubly or triply indirect blocks), the last thirty-two 32-bit words of the block contain a bitmap with a bit for every block referenced in that indirect block.

A similar "block stealing" design was chosen for managing stubs. A number of block addresses in the inode and the indirect blocks have been reserved to point to blocks of stubs. Figure 3.3 illustrates the metadata architecture. The number of direct blocks in an inode has been reduced by one, from twelve to eleven, for storage of stubs (`i_data[11]`) that correspond to the direct blocks. Ext3cow reserves words in indirect blocks to be used as pointers to blocks of stubs.

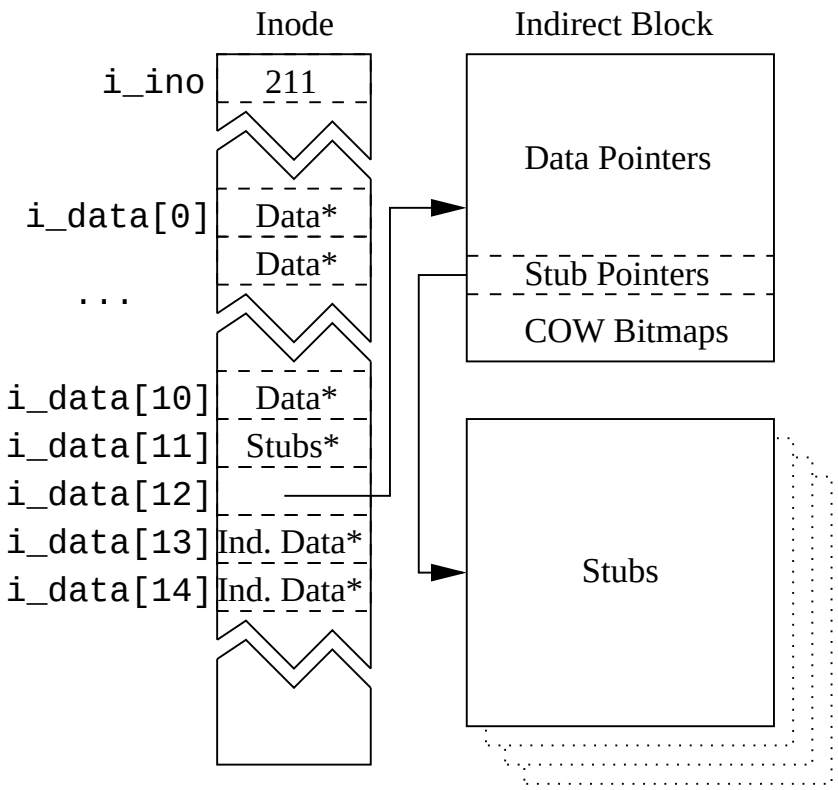


Figure 3.3: Metadata architecture to support stubs.

The number of stub block pointers depends on the file system block size and the encryption method. In AON encryption, four stub blocks are required to hold the stubs corresponding to the 4MB of data described by a 4K indirect block. Because of the message expansion and authentication components of the randomized-key scheme ($c_{n+1}, \ldots, c_m, t$), sixteen stub blocks must be reserved; four for the deletable stubs and twelve for the expansion and authentication. Only the stub blocks must be securely overwritten in order to permanently delete data.

All stub blocks in an indirect block are allocated with strict contiguity. This has two benefits: when securely deleting a file, contiguous stub blocks may be securely overwritten together, improving the time to overwrite. Second, stub blocks may be more easily read when performing an I/O. Stub blocks should not increase the number of I/Os performed by the drive for a read. Ext3cow makes efforts to co-locate data, metadata and stub blocks in a single disk drive track, enabling all to be read in single I/O.

Because the extra metadata borrows space from indirect blocks, the design reduces the maximum file size. The loss is about 16%. With a 4K block size, ext3cow represents files up to 9.03×10^8 blocks in comparison to 1.07×10^9 blocks in ext3. The upcoming adoption of quadruply indirect blocks by ext3 [116] will remove practical file size limitations.

3.4.2 The Secure Block Device Driver

All encryption functionality is contained in a secure block device driver. By encapsulating encryption in a single device driver, algorithms are modular and independent of the file system or other system components. This enables any file system that supports the management of stubs to utilize our device driver.

When encrypting, a data page is passed to the device driver. The driver copies the page into its private memory space, ensuring the user's image of the data is not encrypted. The driver encrypts the private data page, generates a stub, and passes the encrypted page to the low level disk driver. The secure device driver interacts with the file system twice: once to acquire encryption and authentication keys and once to write back the generated stub.

Cryptography in the device driver was built upon the pre-existing cryptographic API available in the Linux kernel [77], namely the AES and SHA-1 algorithms. Building upon existing constructs simplified development, and aids correctness. Further, it allows for the security algorithms to evolve, giving opportunity for the secure deletion transforms to be updated as more secure algorithms become available. For instance, the entropy of SHA-1 has been recently called into question [119].

3.4.3 Security Policies

When building an encrypting, versioning file system, certain policies must be observed to ensure correctness. In our security model, a stub may never be re-written in place once committed to disk.Violating this policy places new stub data over old stub data, allowing the old stub to be recoverable via magnetic force microscopy or other forensic techniques.

With secure deletion, I/O drives the creation of versions. Our architecture mandates a new version whenever a block and a stub are written to disk. Continuous versioning, *e.g.* CVFS [111], meets this requirement, because it creates a new version on every `write()` system call. However, for many users, continuous versioning may incur undesirable storage overheads, approximately 27% [86, 111]. Most systems create versions less frequently. As a matter of policy, *e.g.* daily, on every file open, *etc.*; or, explicitly through a snapshot interface.

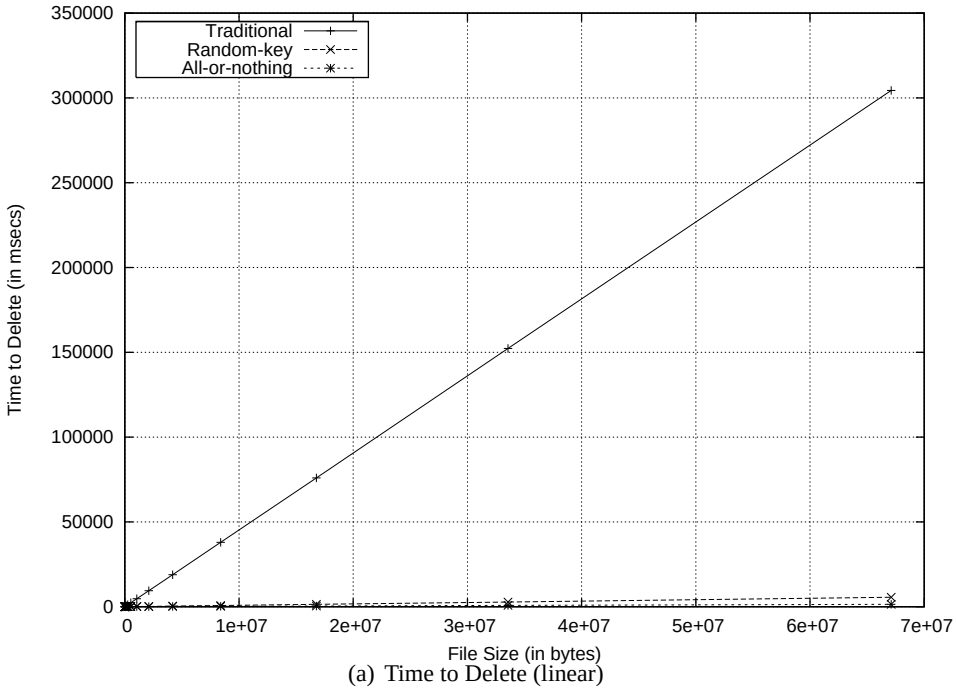
The demands of secure deletion may be met without continuous versioning. Ext3cow reduces the creation of versions based on the observation that multiple writes to the same stub may be aggregated in memory prior to reaching disk. We are developing write-back caching policies that delay writes to stub blocks and aggregate multiple writes to the same stub or writes to multiple stubs within the same disk sector. Stub blocks may be delayed even when the corresponding data blocks are written to disk; data may be re-written without security exposure. A small amount of non-volatile, erasable memory or an erasable journal would be helpful in delaying disk writes when the system call specifies a synchronous write.

3.5 Experimental Results

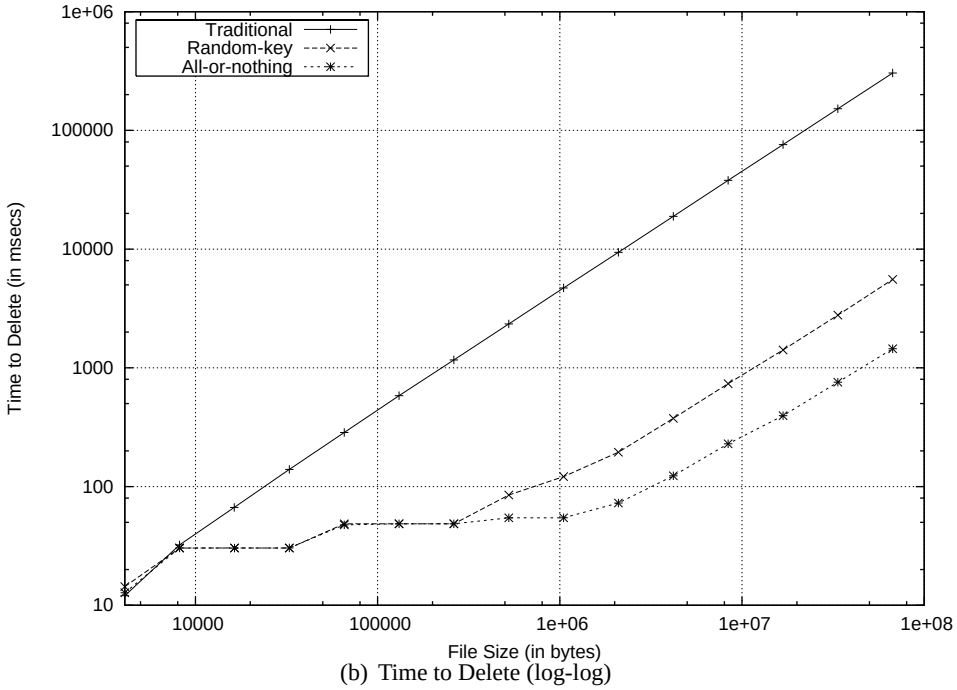
We measure the impact that AON and random-key secure deletion have on performance in a versioning file system. We begin by measuring the performance benefits of deletion achieved by AON and random-key secure deletion. We then use the Bonnie++ benchmark suite to stress the file system under different cryptographic configurations. Lastly, we explore the reasons why secure deletion is a difficult problem for versioning file systems through trace-driven file system aging experiments. All experiments were performed on a Pentium 4, 2.8GHz machine with 1GB of RAM. Bonnie++ was run a 80GB partition of a Seagate Barracuda ST380011A disk drive.

3.5.1 Time to Delete

To examine the performance benefits of our secure deletion techniques, we compared our all-or-nothing and random-key algorithms with Gutmann's traditional secure overwriting technique. Files, sized 2^n blocks for $n = 0, 1, \ldots, 20$, were created; for 4KB blocks, this a file size range of 4KB



(a) Time to Delete (linear)



(b) Time to Delete (log-log)

Figure 3.4: The time to securely delete files for the secure overwriting (traditional), all-or-nothing, and random-key techniques.

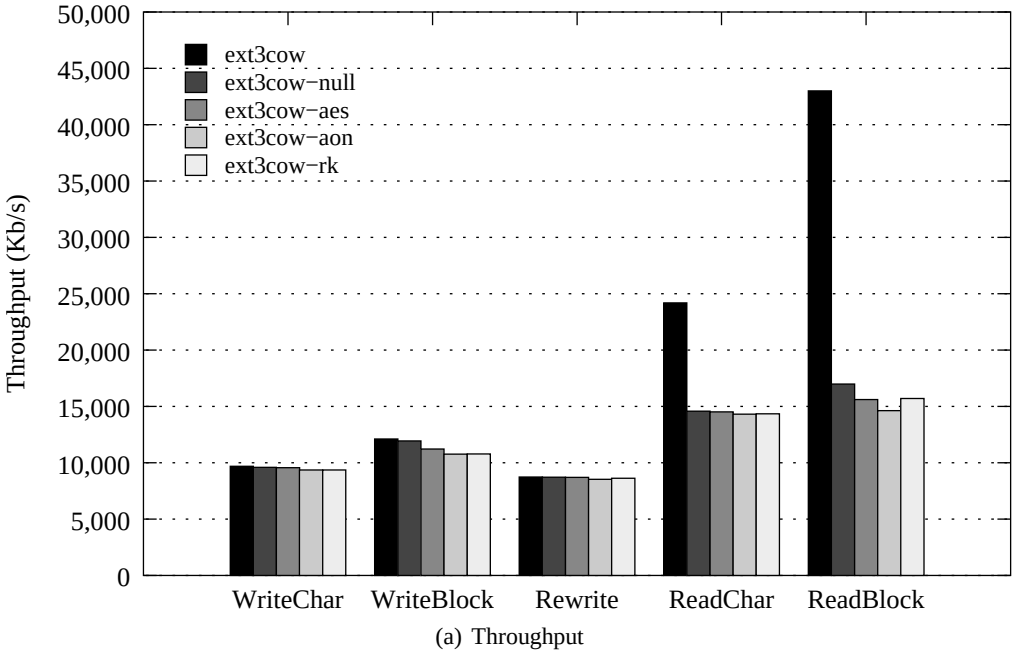
to 4GB. Each file was then securely deleted using each of the three secure deletion methods, and the time to do so was measured. Because no versioning is taking place, files are relatively contiguous on disk. Further, no blocks are shared between versions so all blocks of the file are overwritten.

Figure 3.4(a) demonstrates the dramatic savings in time that can be achieved by using stub deletion. Files between 2^{16} and 2^{20} were truncated for clarity. AON deletion bests traditional deletion by a factor of 200 for 67MB files (2^{15} blocks), with random-key deletion performing slightly worse than AON. Differences are better seen in Figure 3.4(b), a log-log plot of the same result.

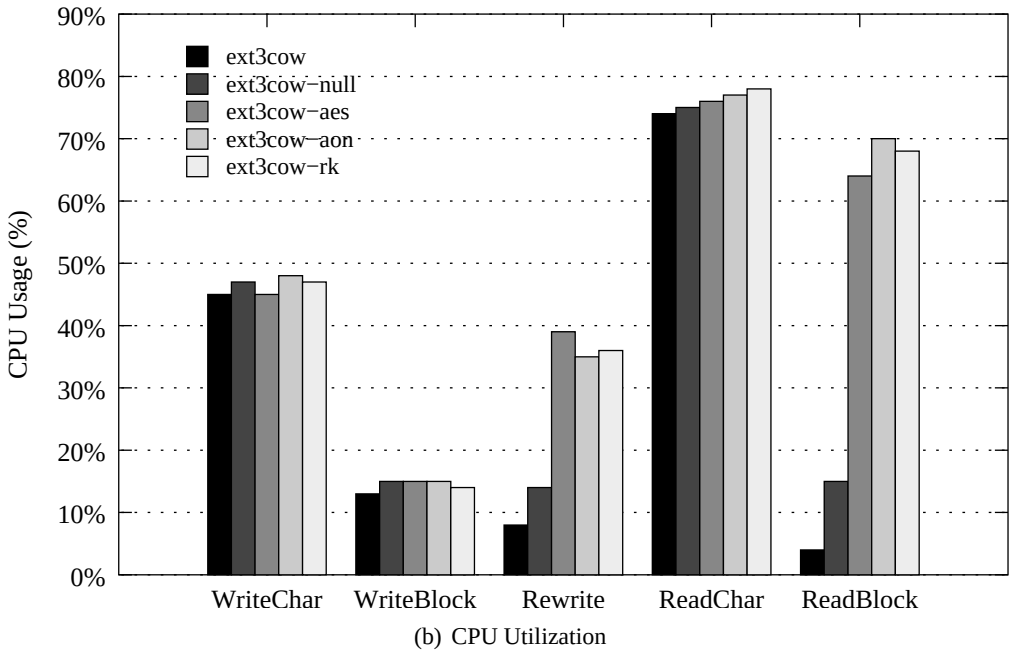
AON and random-key deletion perform similarly on files allocated only with direct blocks (between 2^0 and approximately 2^4 blocks), and begin to diverge at 2^7 blocks. By the time files are allocated using doubly indirect blocks (between 2^9 and 2^{10} blocks) the performance of random-key and AON differ substantially. This is due to the larger stub size needed for random-key deletion, requiring more secure overwriting of stub blocks.

3.5.2 Bonnie++

Bonnie++ is a well-known performance benchmark that quantifies five aspects of file system performance based on observed I/O bottlenecks in a UNIX-based file system. Bonnie++ performs I/O on large files (for our experiment, two 1-GB files) to ensure I/O requests are not served out of the disk's cache. For each test, Bonnie++ reports throughput, measured in kilobytes per second, and CPU utilization, as a percentage of CPU usage. Five operations are tested: (1) each file is written sequentially by character, (2) each file is written sequentially by block, (3) the files are sequentially read and rewritten, (4) the files are read sequentially by character, and (5) the files are read sequentially by block. We compare the results of five file system modes: ext3cow, ext3cow-null, ext3cow-aes, ext3cow-aon and ext3cow-rk. Respectively, they are: a plain installation of



(a) Throughput



(b) CPU Utilization

Figure 3.5: Bonnie++ throughput and CPU utilization results.

ext3cow with no secure device driver. Ext3cow with a secure device driver that does no encryption. Ext3cow with a secure device driver that does a simple AES encryption. Ext3cow with a secure device driver that runs the all-or-nothing algorithm, and ext3cow with a secure device driver that runs the random-key algorithm. Ext3cow performs comparably with ext3 [86]. Results are the product of an average of 10 runs of Bonnie++ on the same partition.

Figure 3.5(a) presents throughput results for each Bonnie++ test. When writing, throughput suffers very little in the presence of cryptography. The largest difference occurs when writing data a block at a time; AON encryption reduces throughput by 1.3 MB/s, from 12.1 MB/s to 10.8 MB/s. This result is consistent with the literature [121]. A more significant penalty is incurred when reading. However, we believe this to be an artifact of the driver and not the cryptography, as the null driver (the secure device driver employing no cryptography) experiences the same performance deficit. The problem stems from the secure device driver's inability to aggregate local block requests into a single large request. We are currently implementing a request clustering algorithm that will eliminate the disparity. In the meantime, the differences in the results for the null device driver and device drivers that employ cryptography are minor: a maximum difference of 200 K/s for character reading and 1.2 MB/s for block reading. Further, the reading of stubs has no effect on the ultimate throughput. We attribute this to ext3cow's ability to co-locate stubs with the data they represent. Because it is based on ext3 [16], ext3cow employs block grouping to keep metadata and data near each other on disk. Thus, track caching on disk and read-ahead in ext3cow put stubs into the disk and system cache, making them readily available when accessing the corresponding data.

To gauge the impact of file system cryptography on the CPU, we measured the CPU utilization for each Bonnie++ test. Results are presented in Figure 3.5(b). When writing, cryptography,

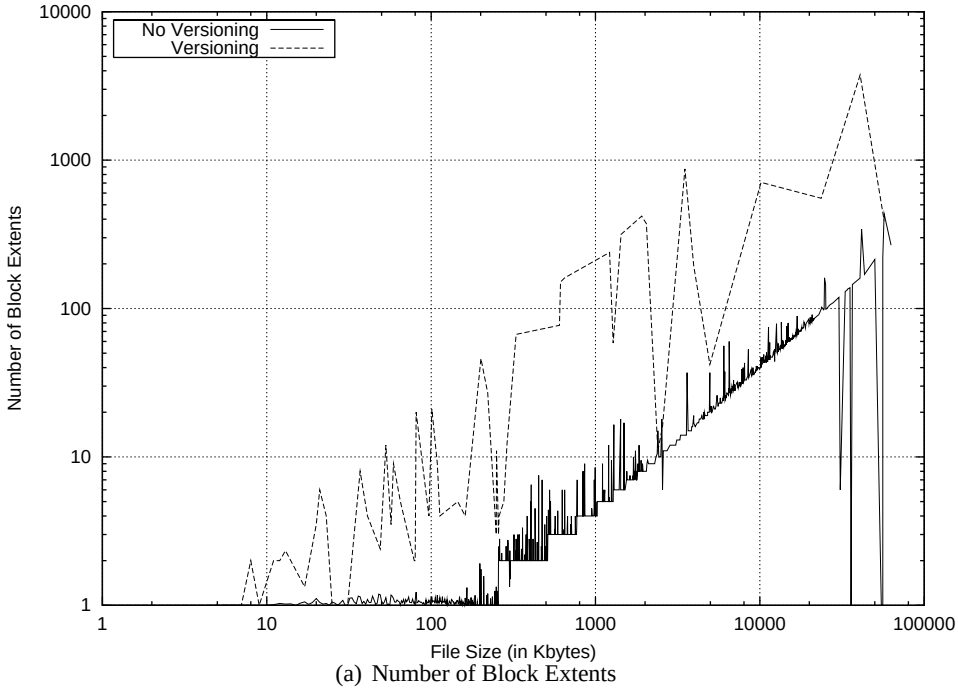
as a percentage of the total CPU, has nearly no effect. This makes sense, as more of the CPU is utilized by the operating system for writing than for reading. Writes may perform multiple memory copies, allocate memory pages, and update metadata. Similarly, reading data character by character is also CPU intensive, due to buffer copying and other memory management operations, so cryptography has a negligible effect. Cryptography does have a noticeable effect when reading data a block at a time, evident in the rewrite and block read experiments. Because blocks match the page size in ext3cow, little time must be spent by the CPU to manage memory. Thus, a larger portion of CPU cycles are spent on decryption. However, during decryption, the system remains I/O bound, as the CPU never reaches capacity. These results are consistent with recent findings [121] that the overheads of cryptography are acceptable in modern file systems.

The cost of cryptography for secure deletion does not outweigh the penalties for falling out of regulatory compliance. In the face of liability for large scale identity theft, the high cost of litigation, and potentially ruinous regulatory penalties, cryptography should be considered a relatively low cost and necessary component of regulatory storage systems.

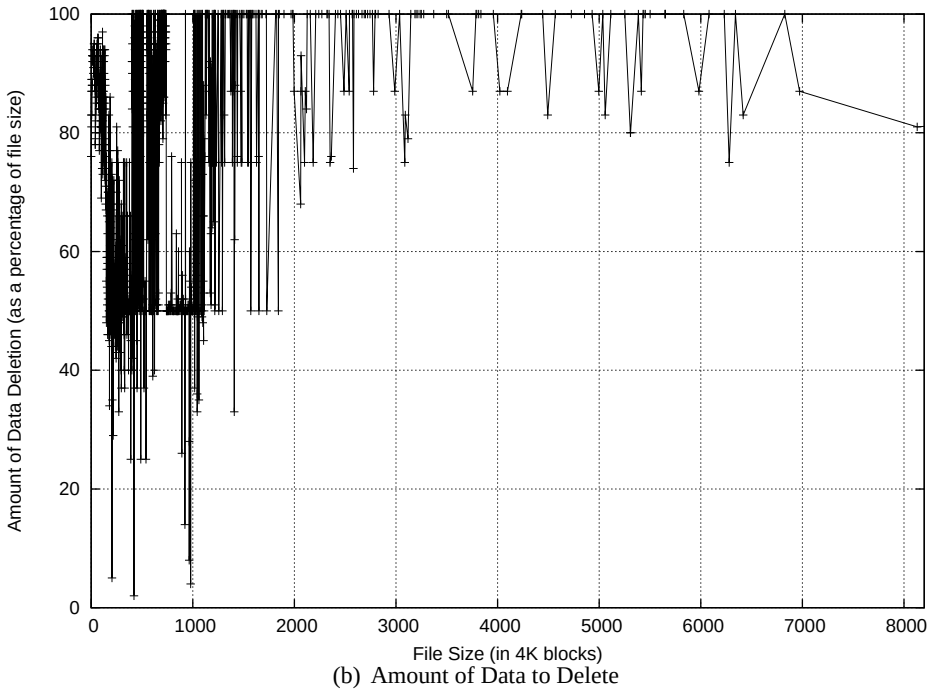
3.5.3 Trace-Driven Experiments

We present results that quantify the difficulty of achieving good performance when securely deleting data that have fallen out of regulatory scope. We replayed four months of file system call traces [97] on an 80G ext3cow partition, taking snapshots every second. This results in 4.2 gigabytes of data in 81,674 files.

We first examine the amount of external fragmentation that results from versioning. External fragmentation is the phenomenon of file blocks in nonadjacent disk addresses. This causes multiple disk drive seeks to read or delete a file. Ext3cow uses a copy-on-write scheme to version



(a) Number of Block Extents



(b) Amount of Data to Delete

Figure 3.6: Results of trace-driven file system aging experiments.

files [86]. This precludes the file system from keeping all blocks of a version strictly contiguous. Because seeks are an expensive operations, fragmentation is detrimental to the performance of traditional secure overwriting. Figure 3.6(a) shows the effect versioning has on block fragmentation. Versioning increases dramatically the average number of block extents – regions of contiguous blocks. This is in comparison to the ext3 file system without versioning. Note the log-log scale. Some files have as many as 1000 block extents. This is the result of files receiving lots copy-on-write versioning.

In practice, secure deletion provides more benefit than microbenchmark results would indicate (Section 3.5.1). Given that seeking is the most expensive disk operation, traditional secure overwriting scales with the number of block extents that need to be overwritten. For AON or random-key secure deletion, the number of extents depends only upon the file size, not the fragmentation of data. Deletion performance does not degrade with versioning. For secure overwriting of the file data, performance scales with the number of block extents. Given the large degree of fragmentation generated through versioning, isolating deletion performance from file contiguity is essential.

Despite the high degree of copy-on-write and fragmentation, trace results show that there are considerable data to delete in each version, *i.e.* deletion is non-trivial. When a version of a file falls out of scope, much of its data are unique to that version and, thus, need to be securely deleted. This is illustrated in Figure 3.6(b). This graph shows the average amount of data that needs to be deleted as a percentage of the file size. There are very few files that have fewer than 25% unique blocks. Most versions need 100% of their blocks deleted. This is not unexpected as many files are written once and never modified. This is much more important for larger files which are more

sensitive to deletion performance; stub deletion offers less benefit when deleting very small files. Even the largest files in the file system contain mostly unique data.

3.6 Applicability to Other Data Systems

There is potential for the reuse of the AON and random-key algorithms for secure deletion in any storage system that shares data among files. Content-indexing systems, such as Venti [91], LBFS [81], and pStore [4], have the same deletion problems and our technology translates directly. Content-indexing stores a corpus of data blocks (for all files) and represents a file as an assemblage of blocks in the corpus. Files that share blocks in the corpus have the same dependencies as do copy-on-write versions.

Chapter 4

Verfiable Audit Trials for a Federally Compliant Storage System

> "History is the version of past events that people have decided to agree upon."
> *– Napoleon Bonaparte*

This chapter presents constructs that create, manage, and verify digital audit trails for versioning file systems. Based upon a small amount of data published to a third party, a file system commits to a version history. At a later date, an auditor uses the published data to verify the contents of the file system at any point in time. Audit trails create an analog of the paper audit process for file data, helping to meet the requirements of electronic record legislation, such as Sarbanes-Oxley. The techniques address the I/O and computational efficiency of generating and verifying audit trails, the aggregation of audit information in directory hierarchies, independence to file system architectures and the construction of verifiable audit trails in the presence of lost data.

4.1 Introduction

The advent of Sarbanes-Oxley (SOX) [21] has irrevocably changed the audit process. SOX mandates the retention of corporate records and audit information. It also requires processes and systems for the verification of the same. Essentially, it demands that auditors and companies

present proof of compliance. SOX also specifies that auditors are responsible for accuracy of the information on which they report. Auditors are taking measures to ensure the veracity of the content of their audit. For example, KPMG employs forensic specialists to investigate the management of information by their clients.

Both auditors and companies require strong audit trails on electronic records; for both parties to prove compliance and for auditors to ensure the accuracy of the information on which they report. The provisions of SOX apply equally to digital systems as they do to paper records. By a "strong" audit trail, we mean a verifiable, persistent record of how and when data have changed.

Current systems for compliance with electronic records legislation meet the record retention and metadata requirements for audit trails, but cannot be used for verification. Technologies such as continuous versioning file systems [111] and temporal databases may be employed in order to construct and query a data history; all changes to data are recorded and the system provides access to the record through time-oriented file system interfaces [86] or through a temporal query language [110]. However, for verification, past versions of data must be immutable. While such systems may prevent writes to past versions by policy, histories may be changed undetectably (see Section 4.3).

The digital audit parallels paper audits in process and incentives. The digital audit is a formal assessment of an organization's compliance with legislation. Specifically, verifying that companies retain data for a mandated period. The audit process does not ensure the accuracy or authenticity of the data itself, nor does it prevent the destruction of data. It verifies that data have been retained, have not been modified, and are accessible within the file system. To fail a digital audit does not prove wrongdoing. Despite its limitations, the audit process has proven itself in the

paper world and offers the same benefits for electronic records. The penalties for failing an audit include fines, imprisonment, and civil liability, as specified by the legislation.

We present a design and implementation of a system for verification of version histories in file systems based on generating message authentication codes (MACs) for versions and archiving them with a third party. A file system commits to a version history when it presents the MAC to the third party. At a later time, a version history may be verified by an auditor. The file system is challenged to produce data that matches the MAC, ensuring that the system's past data have not been altered. Participating in the audit process should reveal nothing about the contents of data. Thus, we consider audit models in which organizations maintain private file systems and publish secure, one-way functions of file data to third parties. Published data may even be stored publicly.

Design goals include minimizing the network, computational, and storage resources used in the publication of data and the audit process. I/O efficiency is the central challenge. We provide techniques that minimize disk I/O when generating audit trails and greatly reduce I/O when verifying past data, when compared with adapting a hierarchy of MACs to versioning systems [40]. We employ parallel message authentication codes (PMAC) [6, 7, 9] that allow MACs to be computed incrementally – based only on data that have changed from the previous version. PMAC generation uses only data written in the cache, avoiding read I/O to file blocks on disk. Sequences of versions may be verified by computing a MAC for one version and incrementally updating the MAC for each additional version, performing the minimum amount of I/O. With incremental computation, a natural trade-off exists between the amount of data published and the efficiency of audits. Data may be published less frequently or on file system aggregates (from blocks into files, files into directories, etc.) at the expense of verifying more data during an audit.

Other technical contributions include a construct for building an audit trail based on hash chaining and constructing hierarchies of audit information in hierarchical namespaces. Additionally, to validate version histories in the presence of failures, we propose the use of approximate MACs (AMAC) [31]. This allows for a weaker statement of authenticity, but supports failure-prone storage environments.

Techniques were designed to be independent of file system architecture. This has two benefits: (1) it allows for verifiable audit trails to be implemented on any file system regardless of architecture, and (2) our design makes the audit robust to disk failures, immune to backup and restore techniques, and allows for integration into information lifecycle management (ILM) systems.

We have implemented incremental authentication in the ext3cow file system, a freely-available, open-source system designed for version management in the regulatory environment [86]. Experimental results show that PMACs can increase performance by 94% under common workloads in a versioning file system when compared to traditional, serial hash MACs (HMAC).

4.2 Related Work

Most closely related to this work is the SFS-RO system [40], which provides authenticity and integrity guarantees for a read-only file system. We follow their model for both the publication of authentication metadata, replicated to storage servers, and use similar hierarchical structures. SFS-RO focuses on reliable and verifiable content distribution; it does not address writes, multiple versions, or efficient constructs for generating MACs.

Recently, there has been some focus on adding integrity and authenticity to storage systems. Oceanstore creates a tree of secure hashes against the fragments of a erasure-coded, dis-

tributed block. This detects corruption without relying on error correction and provides authenticity [120]. Patil *et al* [84] provide a transparent integrity checking service in a stackable file system. The interposed layer constructs and verifies secure checksums on data coming to and from the file system. Haubert *et al* [50] provide a survey of tamper-resistant storage techniques and identify security challenges and technology gaps for multimedia storage systems.

Schneier and Kelsey describe a system for securing logs on untrusted machines [102]. It prevents an attacker from reading past log entries and makes the log impossible to corrupt without detection. They employ a similar "audit model" that focuses on the detection of attacks, rather than prevention. As in our system, future attacks are deterred by legal or financial consequences. While logs are similar to version histories, in that they describe a sequence of changes, the methods in Schneier and Kelsey secure the entire log, *i.e.* all changes to date. They do not authenticate individual changes (versions) separately.

To our knowledge, no previous research has addressed the integrity and authenticity of version sequences with each version individually verifiable, nor devised constructs to update MACs incrementally in a file system.

Efforts at cryptographic file systems and disk encryption are orthogonal to audit trails. Such technologies provide for the privacy of data and authenticate data coming from the disk. However, the guarantees they provide do not extend to a third party and, thus, are not suitable for audit.

4.3 Secure Digital Audits

A digital audit of a versioning file system is the verification of its contents at a specific time in the past. The audit is a challenge-response protocol between an auditor and the file system

to be audited. To prepare for a future audit, a file system generates authentication metadata that commits the file system to its present content. This metadata are published to a third party. To conduct an audit, the auditor accesses the metadata from the third party and then challenges the file system to produce information consistent with that metadata. Using the security constructs we present, passing an audit establishes that the file system has preserved the exact data used to generate authentication metadata in the past. The audit process applies to individual files, sequences of versions, snapshots of directories and directory hierarchies, and an entire file system.

Our general approach resembles that of digital signature and secure time-stamp services, *e.g.* the IETF Time-Stamp Protocol [1]. From a model standpoint, audit trails extend such services to apply to aggregates, containers of multiple files, and to version histories. Such services provide a good example of systems that minimize data transfer and storage for authentication metadata and reveal nothing about the content of data prior to audit. We build our system around message authentication codes, rather than digital signatures, for computational efficiency.

The publishing process requires long-term storage of authenticating metadata with "fidelity"; the security of the system depends on storing and returning the same values. This may be achieved with a trusted third party, similar to a certificate authority. It may also be accomplished via publishing to censorship-resistant stores [118].

The principal attack against which this system defends is the creation of false version histories that pass the audit process. This class of attack includes the creation of false versions – file data that matches published metadata, but differ from the data used in its creation. It also includes the creation of false histories, undetectably inserting or deleting versions into a sequence.

In our audit model, the attacker has complete access to the file system. This includes the ability to modify the contents of the disk arbitrarily. This threat is realistic. For example, disk drives may be accessed directly, through the device interface and on-disk structures are easily examined and modified [39]. In fact, we feel that the most likely attacker is the owner of the file system. For example, a corporation may be motivated to alter or destroy data after it comes under suspicions of malfeasance. The shredding of Enron audit documents at Arthur Anderson in 2001 provides a notable paper analog. Similarly, a hospital or private medical practice might attempt to amend or delete a patient's medical records to hide evidence of malpractice. Such records must be retained in accordance with HIPAA [22].

Obvious methods for securing the file system without a third party are not promising. Disk encryption provides no benefit, because the attacker has access to encryption keys. It is useless to have the file system prevent writes by policy, because the attacker may modify file system code. Write-once, read-many (WORM) stores alone are insufficient, as data may be modified and written to a new WORM device.

Tamper-proof storage devices are a promising technology for the creation of immutable version histories [76]. However, they do not obviate the need for external audit trails, which establish the existence of changed data with a third party. Tamper-resistant storage complements audit trails in that it protects data from destruction or modification. Also, such devices are likely to be expensive and expense is the major obstacle to compliance [48].

4.4 A Secure Version History

The basic construct underlying digital audit trails is a message authentication code (MAC) that authenticates the data of a file version and binds that version to previous versions of the file. We call this a *version authenticator* and compute it on version v_i as

$$A_{v_i} = \mathrm{MAC}_K(v_i||A_{v_{i-1}}); A_{v_0} = \mathrm{MAC}_K(v_0||N) \tag{4.1}$$

in which K is an authentication key and N is a nonce, derived uniquely from file system metadata. N differentiates the authenticators for files that contain the same data, including empty files. The MAC function must be a universal one-way hash function [82]. As a corollary, K must be selected at random by the auditor (Section 4.5.2). By including the version data in the MAC, it authenticates the content of the present version. By including the previous version authenticator, we bind A_{v_i} to a unique version history. This creates a keyed hash chain coupling past versions of the file. The wide application of one-way hash chains in password authentication [63], micropayments [94], and certificate revocation [74] testifies to their utility and security.

The authentication key binds each MAC to a specific identity and audit scope. During an audit, the file system reveals K to the auditor, who may then verify all version histories authenticated with K. K may be securely derived from a known identity, *e.g.* in a public-key infrastructure. In this case, the key binds the version history to that identity. A file system may use many keys to limit the scope of an audit, *e.g.* to a specific user. For example, Plutus supports a unique key for each authentication context [59], called a *filegroup*. Authentication keys derived from filegroup keys would allow each filegroup to be audited independently.

A file system commits to a version history by transmitting and storing version authenticators at a third party. The system relies on the third party to store them persistently and reproduce them accurately, *i.e.* return the stored value keyed by file identifier and version number. It also associates each stored version authenticator with a secure time-stamp [68]. An audit trail consists of a chain of version authenticators and can be used to verify the manner in which the file changed over time. We label the published authenticator P_{v_i}, corresponding to A_{v_i} computed at the file system.

The audit trail may be used to verify the contents of a single version. To audit version v_i, an auditor requests version data v_i and the previous version authenticator $A_{v_{i-1}}$ from the file system, computes A_{v_i} using Equation 4.1 and compares this to the published value P_{v_i}. The computed and published identifiers match if and only if the data currently stored by the file system are identical to the data used to compute the published value. This process verifies the version data content v_i even though $A_{v_{i-1}}$ is untrusted.

We do not require all version authenticators to be published. A version history (sequence of changes) to a file may be audited based on two published version authenticators separated in time. An auditor accesses two version authenticators P_{v_i} and $P_{v_j}, i < j$. The auditor verifies the individual version v_i with the file system. It then enumerates all versions $v_{i+1}, \ldots, v_j$, computing each version identifier in turn until it computes A_{v_j}. Again, A_{v_j} matches P_{v_j} if and only if the data stored on the file system is identical to the data used to generate the version identifiers, *including all intermediate versions*.

Verifying individual versions and version histories relies upon the collision resistance properties of MACs. For individual versions, the auditor uses the untrusted $A_{v_{i-1}}$ from the file system, because the MAC authenticates version v_i even when an adversary can choose input $A_{v_{i-1}}$.

A similar argument allows a version history to be verified based on the authenticators of its first and last version. Finding an alternate version history that matches both endpoints is as difficult as finding a collision.

Version authenticators may be published infrequently. The file system may perform many updates without publication as long as it maintains a local copy of a version authenticator. This creates a natural trade-off between the amount of space and network bandwidth used by the publishing process and the efficiency of verifying version histories.

4.4.1 Incrementally Calculable MACs

I/O efficiency is the principal concern in the calculation and verification of version authenticators at the file system. A version of a file shares data with its predecessor; it differs only the blocks of data that are changed. As a consequence, the file system performs I/O only on these changed blocks. For performance reasons, it is imperative that the system updates audit trails based only on the changed data.

To achieve our efficiency goals, we employ a parallel message authentication code (PMAC) [6, 7, 9] to compute version authenticators. 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 effort to compute the authenticator scales with the size of the changed data, and, thus, with the amount of I/O. In contrast, a serial MAC requires the whole file to be examined in the construction of the MAC. A PMAC is a MAC and, thus, preserves all of its security properties [9].

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. Each version v_i comprises blocks $b_{v_i}(0), \ldots, b_{v_i}(n)$ equal to the file system block size and a file system independent representation of the versions metadata, denoted $\overline{M}_{v_i}$. To be consistent with the original publication [9], for block b_{v_i}, we label the one-way function on each block $Y(b_{v_i})$. The output of the PMAC is the exclusive-or of the one-way functions of the input blocks and the previous version authenticator.

$$A_{v_i} = \bigotimes_{j=0}^{n} Y(b_{v_i}(j)) \otimes Y(A_{v_{i-1}}) \otimes Y(\overline{M}_{v_i}). \tag{4.2}$$

This form is the full computation. There is also an incremental computation. Assuming that version v_i differs from v_{i-1} in one block only, *e.g.* $b_{v_i}(j) = b_{v_{i-1}}(j), j \neq k; b_{v_i}(k) \neq b_{v_{i-1}}(k)$, we observe that

$$A_{v_i} = A_{v_{i-1}} \otimes Y(b_{v_i}(k)) \otimes Y(b_{v_{i-1}}(k)) \otimes Y(A_{v_{i-2}}) \otimes Y(A_{v_{i-1}}) \otimes Y(\overline{M}_{v_i-1}) \otimes Y(\overline{M}_{v_i}).$$

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. It also updates the past version authenticator and metadata.

The computation of PMAC authenticators scales with I/O size whereas the performance of a hash message authentication code (HMAC) scales with the file size. With PMACs, only new data being written to a version will be authenticated. HMACs must authenticate the entire file, irrespective of the I/O size. This is problematic as studies of versioning file systems show that data change at a fine granularity [86, 111]. Our results (Section 4.6) confirm the same. More importantly, the computation of the updated PMAC version authenticator may be performed on data available in the cache, requiring little to no additional disk I/O. PMAC computations require only

those data blocks being modified, which are already in cache. Computing an HMAC may require additional I/O. This is because system caches are managed on a page basis, leaving unmodified and inaccessible portions of an individual file version on disk. When computing an HMAC for a file, all file data would need to be accessed. As disk accesses are a factor of 10^5 slower than memory accesses, computing an HMAC may be substantially worse than algorithmic performance would indicate.

The benefits of incremental computation of MACs apply to both writing data and conducting audits. When versions of a file share much data in common, the differences between versions are small, allowing for efficient version verification. Incremental MACs allow an auditor to authenticate the next version by computing the authenticity of only the data blocks that have changed. When performing an audit, the authenticity of the entire version history may be determined by a series of small, incremental computations. HMACs do not share this advantage and must authenticate all data in all versions to verify authenticity.

4.4.2 File System Independence

Many storage management tasks alter a file system, including the metadata of past versions, but should not result in an audit failure. Examples include: file-oriented restore of backed-up data after a disk failure, resizing or changing the logical volumes underlying a file system, compaction/defragmentation of storage, and migration of data from one file system to another. Thus, audit models must be robust to such changes. We call this property *file system independence*. Audit information is bound to the file data and metadata, transfers from system to system, and remains valid when the physical implementation of a file changes with the caveat that all systems storing data support audit trails.

Our authenticators use the concept of *normalized metadata* for file system independence. Normalized metadata are the persistent information that describe attributes of a file system object independent of the file system architecture. These metadata include: file size, ownership and permissions, and modification, creation and access times. These fields are common to most file systems and are stored persistently with every file. Normalized metadata do not include physical offsets and file system specific information, such as inode number, disk block addresses, or file system flags. These fields are volatile in that storage management tasks change their values. Normalized metadata are included in authenticators and become. part of a file's data for the purposes of audit trails.

4.4.3 Hierarchies and File Systems

Audit trails must include information about the entire state of the file system at a given point in time. Auditors need to discover the relationships between files and interrogate the contents of the file system. Having found a file of interest in an audit, natural questions include: what other data was in the same directory at this time? or, did other files in the system store information on the same topic? The data from each version must be associated with a coherent view of the entire file system.

Authenticating directory versions as if they were file versions is insufficient. A directory is a type of file in which the data are directory entries (name-inode number pairs) used for indexing and naming files. Were we to use our previous authenticator construction (Equation 4.2), a directory authenticator would be the MAC of its data (directory entries), the MAC of the previous directory authenticator and its normalized metadata. However, this construct fails to bind the data of a directory's files to the names, allowing an attacker to undetectably exchange names of files within a directory.

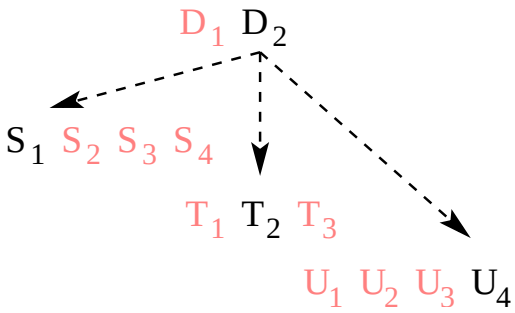
We have developed a construction of trees of MACs that bind individual versions and their names to a file system hierarchy, which authenticates the entire versioning file system. In addition to the normalized inode information and previous authenticator used to authenticate files, directory authenticators are composed of name-authenticator pairs. Each file within the directory concatenates its authenticator to its corresponding name, and a one-way hash of the result is taken.

$$A_{D_i} = \bigotimes_{j=0}^{n} Y(name_j | A_{v_j}) \otimes Y(A_{D_{i-1}}) \otimes Y(\overline{M}_{D_i}).$$

This binds each file and sub-directories data to their names in the parent directory. Directory version authenticators continue recursively to the file system root, binding the entire file system image. The SFS-RO system [40] employed a similar technique to fix the content of a read-only file system with single versions of each file and directory. Our methods differ from SFS-RO in that they are incremental and must account for updates.

For efficiency reasons, we bind versions to the directory's authenticator lazily. Figure 4.1 shows how directory D binds to files S, T, U. This is done by including the authenticators for specific versions S_1, T_2, U_4 that were current at the time version D_2 was created. However, subsequent file versions (*e.g.* S_2, T_3) may be created without updating the directory version authenticator A_{D_2}. The system updates the directory authenticator only when the contents change; *i.e.* files are created, destroyed or renamed. This corresponds well with our notion of a directory version. In this example, when deleting file U (Figure 4.1) the authenticator is updated to the current versions. Were we to bind directory version authenticators directly to the content of the most recent file version, they would need to be updated every time that a file is written. This includes all parent directories recursively to the file system root – an obvious performance concern.

$$A_{D_2} = Y(S|A_{S_1}) \otimes Y(T|A_{T_2}) \otimes Y(U|A_{U_4}) \otimes Y(A_{D_1}) \otimes Y(\overline{M}_{D_2})$$



$$A_{D_3} = Y(S|A_{S_4}) \otimes Y(T|A_{T_5}) \otimes Y(A_{D_2}) \otimes Y(\overline{M}_{D_3})$$

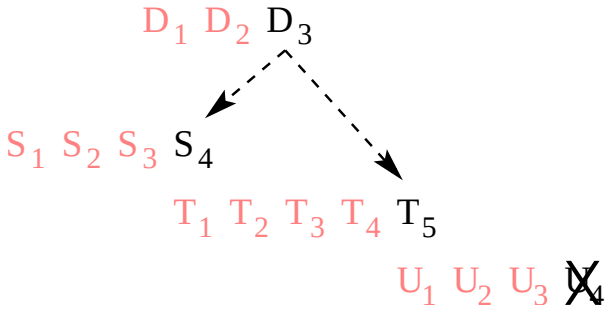


Figure 4.1: Updating directory version authenticators when file U is deleted.

Binding a directory authenticator to a file version binds it to all subsequent versions of that file, by hash chaining of the file versions. This is limited to the portion of the file's version chain within the scope of the directory. (Ext3cow employs timestamps for version numbers, which can be used to identify the valid file versions within each directory version.)

Updating directory authenticators creates a time-space trade-off similar to that of publication frequency (Section 4.4). When auditing a directory at a particular point in time, the auditor must access the directory at a point in time and then follow the children files' hash chains forward to that point in time. Thus, updating directory authenticators more frequently may be desirable to speed the audit process.

4.5 File System Implementation

We have implemented digital audit trails using PMACs in ext3cow [86], an open-source, block-versioning file system designed to meet the requirements of electronic record management legislation. Versions of a file are implemented by chaining inodes together in which each inode represents a version. The file system traverses the inode chain to generate a point-in-time view of a file. Ext3cow provides the features needed for an implementation of audit trails: it supports continuous versioning, creating a new version on every write; and, maintains old and new versions of data and metadata concurrently for the incremental computation of version authenticators using parallel MACs. Version authentication is achieved by storing that version's MAC in its corresponding inode. We have already retrofitted the metadata structures of ext3 to support versioning and secure deletion based on authenticated encryption [87, 88].Version authenticators are a straightforward extension to ext3cow's already augmented metadata, requiring only a few bytes per inode.

4.5.1 Metadata for Authentication

Metadata in ext3cow have been improved to support incremental versioning authenticators for electronic audit trails. To accomplish this, ext3cow "steals" a single data block pointer from the inode, replacing it with an authentication block pointer, *i.e.* a pointer to disk block holding authentication information. Figure 4.2 illustrates the metadata architecture. The number of direct blocks has been reduced by one, from twelve to eleven, for storing an authenticator block (`i_data[11]`). Block stealing for authenticators reduces the effective file size by only one file system block, typically 4K.

Each authenticator block stores five fields: the current version authenticator (A_{v_i}), the authenticator for the previous version ($A_{v_{i-1}}$), the one-way hash of the authenticator for the previous version ($Y(A_{v_{i-1}})$), the authenticator for the penult-previous version ($A_{v_{i-2}}$), and the the one-way hash of the authenticator for the penult-previous version ($Y(A_{v_{i-2}})$). Each current authenticator computation requires access to the previous and penult-previous authenticators and their hashes. By storing authenticators and hashes for previous versions with the current version, the system may avoid two read I/Os, one for each previous version authenticator and hash computations. When a new version is generated and a new inode is created, the authenticator block is copy-on-written and "bumps" each entry; *i.e.*, copying the once current authenticator (A_{v_i}) to the previous authenticator ($A_{v_{i-1}}$), and the previous authenticator ($A_{v_{i-1}}$) and hash ($Y(A_{v_{i-1}})$) to the penult-previous authenticator ($A_{v_{i-2}}$) and hash ($Y(A_{v_{i-2}})$). Since the hash of the once current authenticator (A_{v_i}) is not yet known, the $Y(A_{v_{i-1}})$ field is zeroed, and is calculated on a as-needed basis.

Authenticator blocks should not increase the number of I/Os performed by the system. The block allocator in ext3cow makes efforts to co-locate data, metadata and authenticator blocks

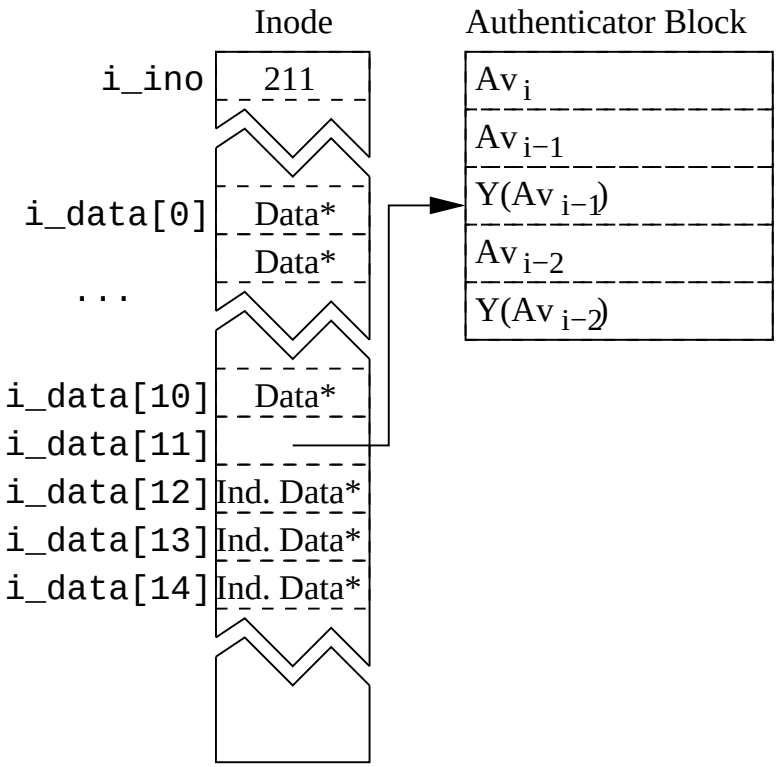


Figure 4.2: Metadata architecture to support version authenticators.

in a single disk drive track, maintaining contiguity. Authenticator blocks are very likely to be read in the same I/O as the inode or data blocks, allowing the authenticator block to be read out of the track cache.

4.5.2 Key Management

Key management in ext3cow uses lockboxes [59] to store a per-file authentication key. The file owner's private key unlocks the lockbox and provides access to the authentication key. Lockboxes were developed as part of the authenticated encryption and secure deletion features of ext3cow [87, 88].

Per-file authentication keys are generated by the system in collaboration with the auditor. The auditor must add randomness to the generation of K to meet the definition of a universal one-way hash function [82]. Authentication keys are then stored within within a user's lockbox.

4.6 Experimental Results

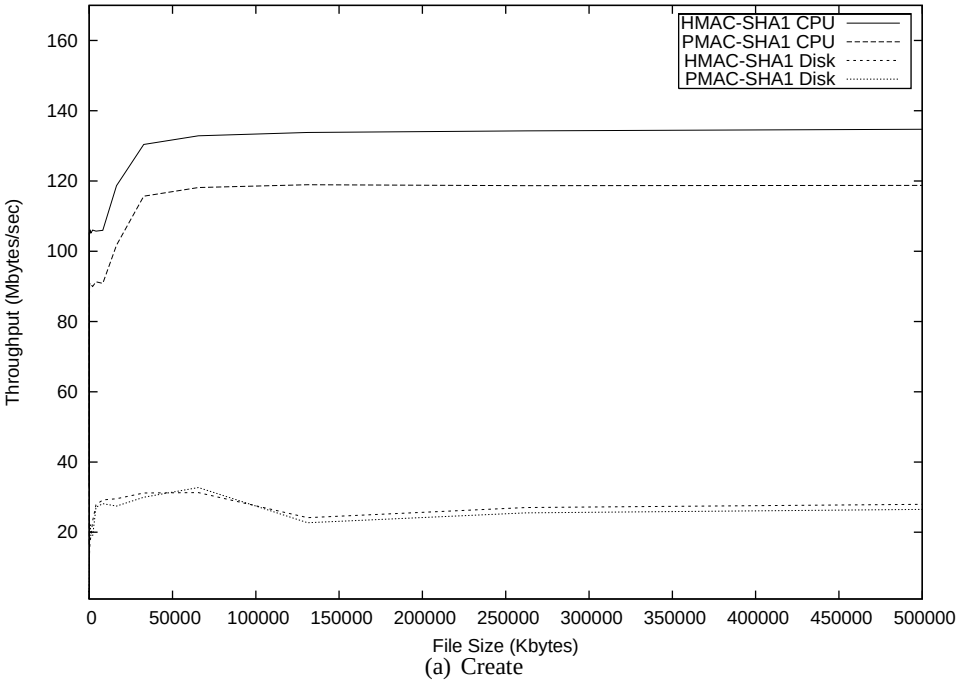
We measure the impact of authentication on versioning file systems and compare the performance characteristics of HMAC and PMAC in the ext3cow versioning file system. We begin by comparing the CPU and disk throughput performance of HMAC and PMAC by using two micro-benchmarks; one designed to contrast the maximum throughput capabilities of each algorithm, and one designed to highlight the benefits of the incremental properties of PMAC. We then use a traced file system workload to illustrate the aggregate performance benefits of incremental authentication in a versioning file system. Lastly, we use file system traces to characterize some of the overheads of generating authenticators for the auditing environment. Both authentication functions, PMAC

and HMAC, were implemented in the ext3cow file system using the standard SHA1 hash function provided by the Linux kernel cryptographic API [77]. All experiments were performed on a Pentium 4, 2.8GHz machine with 1 gigabyte of RAM. Trace experiments were run on a 80 gigabyte ext3cow partition of a Seagate Barracuda ST380011A disk drive.

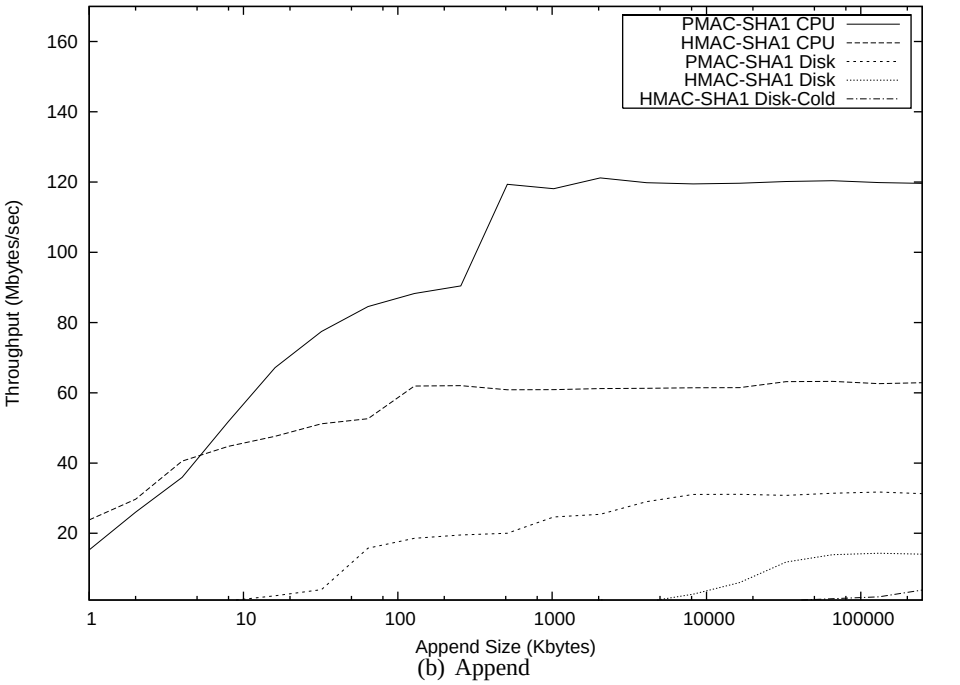
4.6.1 Micro-benchmarks

To quantify the efficiency of PMAC, we conducted two micro-benchmark experiments: *create* and *append*. The *create* test measures the throughput of creating and authenticating files of size 2^N bytes, where $N = 0, 1, \ldots, 30$ (1 byte to 1 gigabyte files). The test measures both CPU throughput, *i.e.* the time to calculate a MAC, and disk throughput, *i.e.* the time to calculate a MAC and write the file to disk. Files are created and written in their entirety. Thus, there are no benefits from incremental authentication. The *append* experiment measures the CPU and disk throughput of appending 2^N bytes to the same file and calculating a MAC, where $N = 0, 1, \ldots, 29$ (1 byte to 500 megabytes). For PMAC, an append requires only a MAC of each new data block and an XOR of the results with the file's authenticator. HMAC does not have this incremental property and must MAC the entire file in order to generate the correct authenticator, requiring additional read I/O. We measure both warm and cold cache configurations. In a warm cache, previous appends are still in memory and the read occurs at memory speed. In practice, a system does not always find all data in cache. Therefore, the experiment was also run with a cold cache; before each append measurement, the cache was flushed.

Figure 4.3(a) presents the results of the *create* micro-benchmark. Traditional HMAC-SHA1 has higher CPU throughput than PMAC-SHA1, saturating the CPU at 134.8 MB/s. The PMAC achieves 118.7 MB/s at saturation. This is expected, as PMAC-SHA1 must perform at least



(a) Create



(b) Append

Figure 4.3: Results of micro-benchmarks measuring the CPU and disk throughput.

one extra call to SHA1 for each block [6]. Additionally, SHA1 appends the length of the message that it's hashing to the end of the message, padding up to 512-bit boundaries. PMAC-SHA1, therefore, hashes more data; up to *n**512 bits more for *n* blocks. Despite PMAC's computational handicap, disk throughput measurements have less disparity. HMAC-SHA1 achieves a maximum of 28.1 MB/s and PMAC-SHA1 a maximum of 26.6 MB/s. This illustrates that calculating authenticators for a file system is I/O-bound, making PMAC-SHA1's ultimate performance comparable to HMAC-SHA1.

The results of the *append* micro-benchmark makes a compelling performance argument and are exhibited in Figure 4.3(b) – note the log scale. We observe PMAC-SHA1 outperforms HMAC-SHA1 in both CPU and disk throughput measurements. PMAC-SHA1 bests HMAC-SHA1 CPU throughput, saturating at 120.3 MB/s, compared to HMAC-SHA1 at 62.8 MB/s. Looking at disk throughput, PMAC-SHA1 also outperforms the best-case of an HMAC calculation, warm-cache HMAC-SHA1, achieving a maximum 31.7 MB/s, compared to warm-cache HMAC-SHA1 at 20.9 MB/s and cold-cache HMAC-SHA1 at 9.7 MB/s. This performance gain is a function of the incremental nature of PMACs. In addition to the extra computation to generate the MAC, an ancillary read I/O is required to bring the old data into the MAC buffer. While the *append* benchmark is contrived, it is a common I/O pattern. Many versioning file systems implement versioning with a copy-on-write policy. Therefore, all I/O that is not a full overwrite is, by definition, incremental and benefits from the incremental qualities of PMAC.

4.6.2 Aggregate Performance

We take a broader view of performance by quantifying the aggregate benefits of PMAC on a versioning file system. To accomplish this, we replayed four months of system call traces [97] on

No Authentication	PMAC-SHA1	HMAC-SHA1
1.98 MB/s	1.77 MB/s	108.78 KB/s

Table 4.1: The trace-driven throughput of no authentication, PMAC-SHA1, and HMAC-SHA1.

an 80 gigabyte ext3cow partition, resulting in 4.2 gigabytes of data in 81,674 files. Our experiments compare trace-driven throughput performance as well as the total computation costs for performing a digital audit using the PMAC and HMAC algorithms. We analyze aggregate results of run-time and audit performance and examine how the incremental computation of MACs benefits copy-on-write versioning.

Write Performance

The incremental computation of PMAC minimally degrades on-line system performance as compared to HMAC. We measure the average throughput of the system while replaying four months of system call traces. The traces were played as fast as possible in an effort to saturate the I/O system. The experiment was performed on ext3cow using no authentication, HMAC-SHA1 authentication and PMAC-SHA1 authentication. Results are presented in Table 4.1. PMAC-SHA1 is able to achieve a 93.9% improvement in run-time performance over HMAC-SHA1; 1.77 MB/s versus 108.78 KB/s. HMAC-SHA1's degradation is due to the additional read I/O and computation time it must perform for every call to write. The performance penalty incurred by PMAC-SHA1 is minimal due to its ability to compute authenticators using only in-cache data. PMAC-SHA1 achieves 89% of the throughput of a system with no authentication.

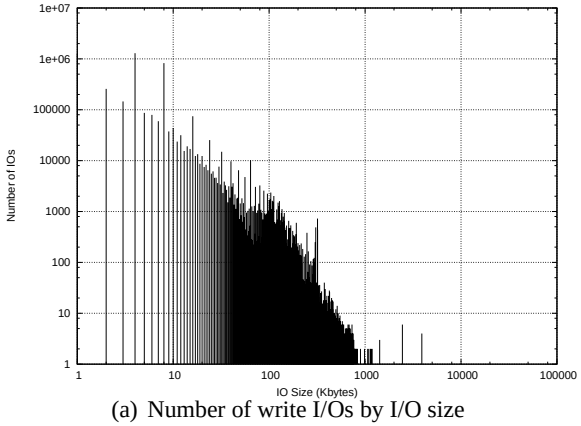
To better understand the run-time performance differences between PMAC and HMAC, we characterize the number and size of writes and how they are written to the various files in the

system. Figure 4.4(a) presents statistics on the number and size of write I/Os, whereas Figure 4.4(b) shows number of write I/Os performed by file size. Both plots are log-log. We observe that of the 16,601,128 write I/Os traced over four months, 99.8% of the I/Os are less than 100K, 96.8% are less than 10K, and 72.4% are less than 1K in size. This elucidates the fact that a substantial number of I/Os are small. We also observe that files of all sizes receive many writes. Files as large as 100 megabytes receive as many as 37,000 write I/Os over the course of four months. Some files, around 5MB in size, receive nearly two million I/Os. These graphs show that I/O sizes are, in general, small, and that files of all sizes receive many I/Os.

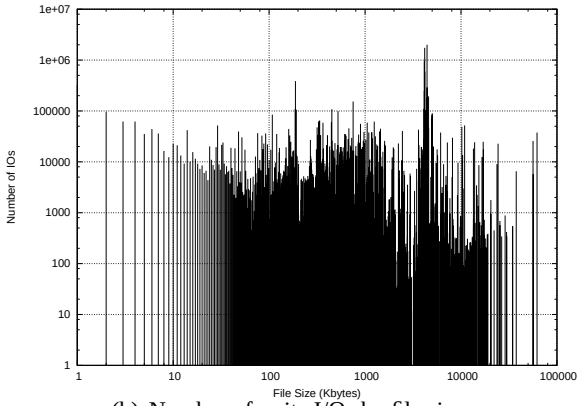
The relationship between I/O size and file size reveals the necessity of incremental MAC computation. Figure 4.4(c) presents the average write I/O size as a percentage of the file size over file sizes. This plot shows that there are few files that receive large writes or entire overwrites in a single I/O. In particular, files larger than 2MB receive writes that are a very small percentage of their file size. The largest files receive as little as 0.025% of their file size in writes, and nearly all files receive less than 25% of their file size in write I/Os. It is this disproportionate I/O pattern that benefits the incremental properties of PMAC. When most I/Os received by large files are small, a traditional HMAC suffers in face of additional computation time and supplementary I/Os. The performance of PMAC, however, is immune to file size and is a function of write size alone.

Audit Performance

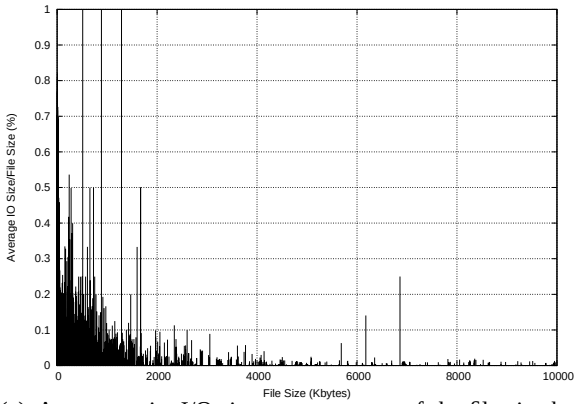
To generate aggregate statistics for auditing, we aged the file system by replaying four months of traced system calls, taking snapshots daily. We then performed two audits of the file system, one using HMAC-SHA1 and one using PMAC-SHA1. Our audit calculated authenticators for every version of every file. Table 4.2 presents the aggregate results for performing an audit using



(a) Number of write I/Os by I/O size



(b) Number of write I/Os by file size



(c) Average write I/O size as percentage of the file size by file size

Figure 4.4: Characterization of write I/Os from trace-driven experiments.

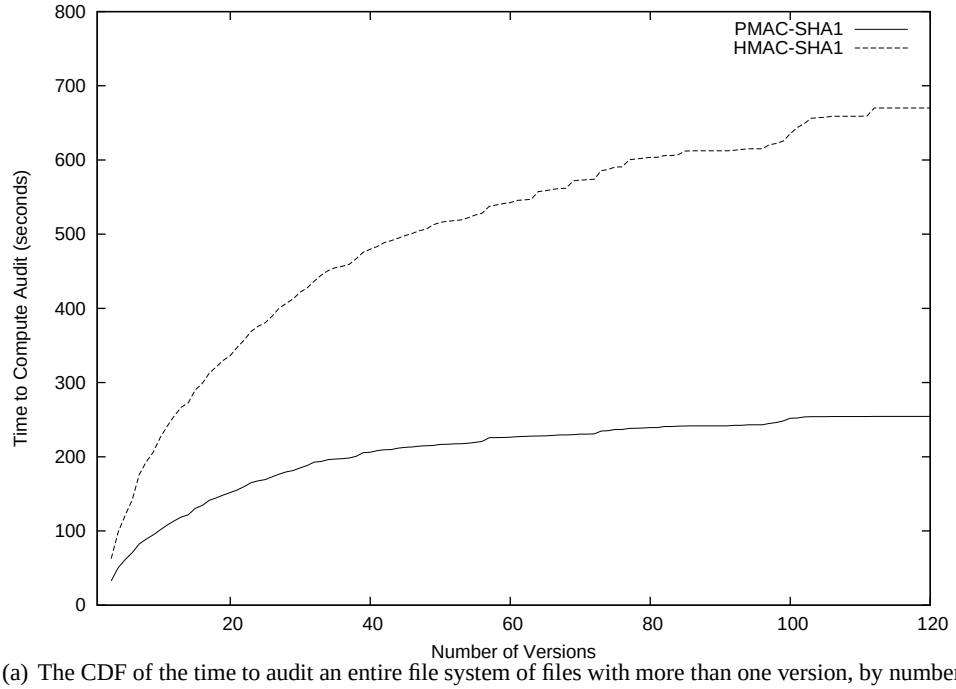
Number of Versions	HMAC-SHA1 (seconds)	PMAC-SHA1 (seconds)
All	11209.4	10593.1
≥ 2	670.1	254.4

Table 4.2: The number of seconds required to audit all files and files with two or more version in an entire file system using HMAC-SHA1 and PMAC-SHA1.

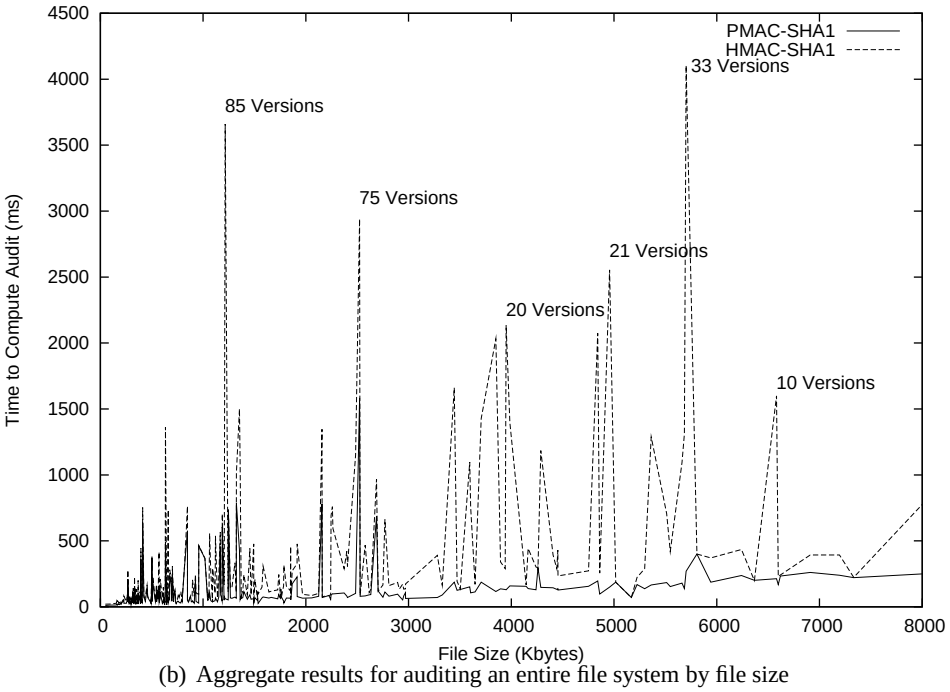
PMAC-SHA1 and HMAC-SHA1. The table shows the result for all files and the result for those files with two or more versions. Auditing the entire 4.2 gigabytes of file system data using standard HMAC-SHA1 techniques took 11,209 seconds, or 3.11 hours. Using PMAC-SHA1, the audit took 10,593 seconds, or 2.94 hours; a savings of 5% (10 minutes).

Most files in the trace (88%) contain a single version, typical of user file systems. These files dominate audit performance and account for the similarity of HMAC and PMAC results. However, we are interested in file systems that contain medical, financial, and government records and, thus, will be populated with versioned data. To look auditing performance in the presence of versions, we filter out files with only one version. On files with two or more versions, PMAC-SHA1 achieves a 62% performance benefit over HMAC-SHA1, 670 versus 254 seconds. A CDF of the time to audit files by number of versions is presented in Figure 4.5(a). PMAC-SHA1 achieves a range of 37% to 62% benefit in computation time over HMAC-SHA1 for files with 2 to 112 versions. This demonstrates the power of incremental MACs when verifying long version chains. The longer the version chain and the more data in common, the better PMAC performs.

Looking at audit performance by file size shows that the benefit is derived from long version chains. Figure 4.5(b) presents a break down of the aggregate audit results by file size. There exists no points at which PMAC-SHA1 performs worse than HMAC-SHA1, only points where they are the same or better. Where PMAC-SHA1 and HMAC-SHA1 points intersect, files either have



(a) The CDF of the time to audit an entire file system of files with more than one version, by number of versions



(b) Aggregate results for auditing an entire file system by file size

Figure 4.5: Aggregate auditing performance results for PMAC-SHA1 and HMAC-SHA1.

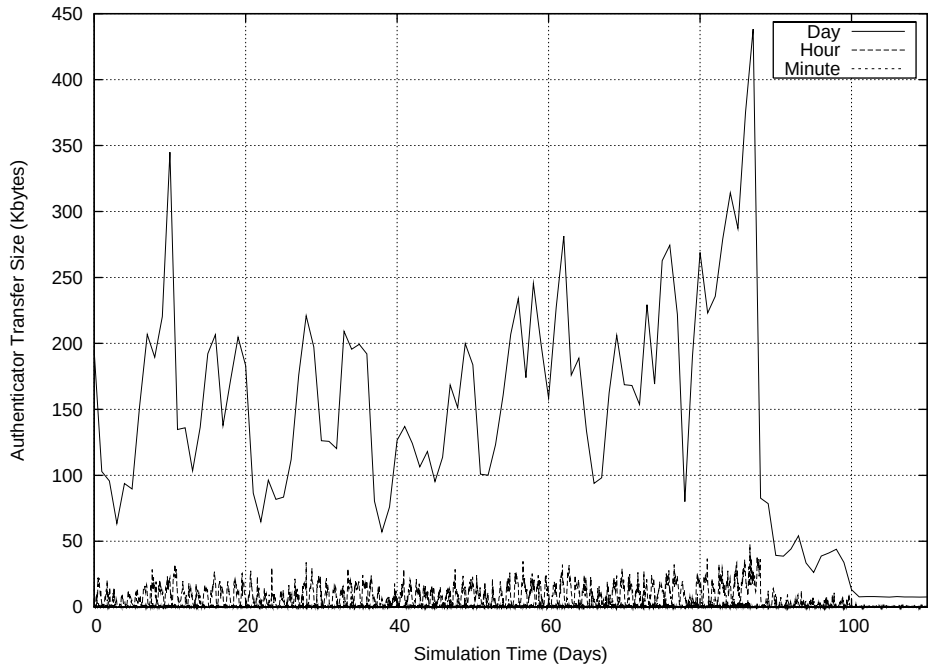


Figure 4.6: Size of authentication data from four months of traced workloads at three snapshot intervals.

a single version or are files in which versions contains no shared data. As the number of versions increase and much data are shared between versions, large discrepancies in performance arise. Some examples of files with many versions that share data are annotated. PMAC shows little performance variance with the number of versions.

4.6.3 Requirements for Auditing

As part of our audit model, authenticators are transfered to and stored at a third party. We explore the storage and bandwidth resources that are required for version authentication. Four months of file system traces were replayed over different snapshot intervals. At a snapshot, authentication data is transfered to the third party, committing the file system to that version history. Measurements were taken at day, hour and minute snapshot intervals. During each interval, the number of file modification and number of authenticators generated were captured.

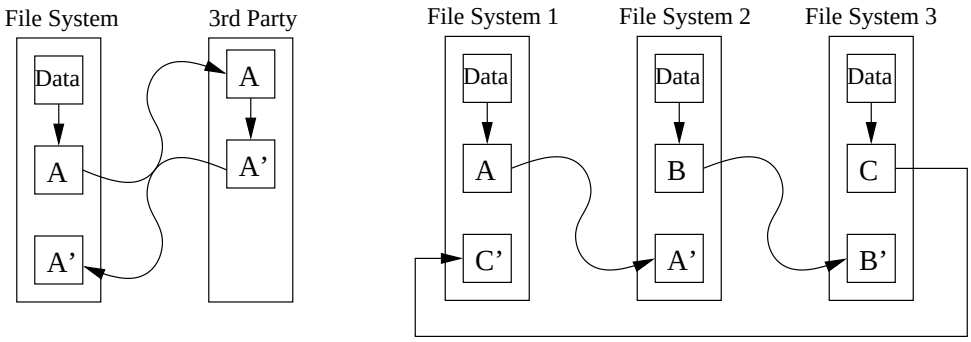
Figure 4.6 presents the size of authentication data generated over the simulation time for the three snapshot intervals. Naturally, the longer the snapshot interval, the larger the number of authenticators generated. However, authentication data is relatively small; even on a daily snapshot interval, the largest transfer is 450K, representing about 22,000 modified files. Authenticators generated by more frequent snapshot (hourly or per-minute) never exceed 50KB per transfer. Over the course of four months, a total of 15.7MB of authentication data is generated on a daily basis from 801,473 modified files, 22.7MB on a hourly basis from 1,161,105 modified files, and 45.4MB on a per-minute basis from 2,324,285 modified files. The size of authenticator transfer is invariant of individual file size or total file system size; it is directly proportional to the number of file modifications made in a snapshot interval. Therefore, the curves in Figure 4.6 are identical to a figure graphing the number of files modified over the same snapshot intervals.

4.7 Future Work

Conducting digital audits with version authenticators leaves work to be explored. We are investigating authentication and auditing models that do not rely on trusted third parties. We also discuss an entirely different model for authentication based on approximate MACs, which can tolerate partial data loss.

4.7.1 Alternative Authentication Models

Having a third party time-stamp and store a file system's authenticators may place undue burden, in terms of storage capacity and management, on the third party. Fortunately, it is only one possible model for a digital auditing system. We are currently exploring two other possible archi-



(a) Storage-less Third Party (b) Cooperative Authentication

Figure 4.7: Alternative models for digital auditing.

tectures for managing authentication data; a storage-less third party and cooperative authentication. In a storage-less third party model (Figure 4.7(a)) a file system would generate authenticators and transmit them to a third party. Instead of storing them, the third party would MAC the authenticators and return them to the file system. The file system stores both the original authenticators and those authenticated by the third party. In this way, the third party stores nothing but signing keys, placing the burden of authentication storage completely on the file system. When the file system is audited, the auditor requests the signing keys from the third party, and performs two authentication steps: first, checking the legitimacy of the stored authenticators and then checking the authenticity of the data themselves.

This design has limitations. The scheme doubles the amount of authentication data transfered. Additionally, because the third party keeps no record of any file, an attacker may delete an entire file system without detection or maintain multiple file systems, choosing which file system to present at audit time. Portions of the file system may not be deleted or modified, because the authenticators for version chains and directory hierarchies bind all data to the root authenticator.

A further variant (Figure 4.7(b)) groups peers of file systems together into a cooperative

ring, each storing their authentication data on an adjoining file system. A file system would store the previous system's authenticator in a log file, which is subsequently treated as data, resulting in the authenticators being authenticated themselves. This authenticator for the log file is stored on an adjoining system, creating a ring of authentication. This design relieves the burden on a single third party from managing all authentication data and removes the single point of failure for the system. This architecture also increases the complexity of tampering by a factor of N, the number of links of in the chain. In order to tamper with a file, an attacker must undetectably modify the data, the data's authenticator, the authenticator's authenticator, and so on. Because an adjoining file system's authenticators are kept in a single log file, only one authenticator is generated for that entire file system, preventing a glut of authentication data.

4.7.2 Availability and Security

A verifiable file system may benefit from accessing only a portion of the data to establish authenticity. Storage may be distributed across unreliable sites [32, 62], such that accessing it in entirety is difficult or impossible. Also, if data from any portion of the file system are corrupted irreparably, the file system may still be authenticated, whereas with standard authentication, altering a single bit of the input data leads to a verification failure.

To audit incomplete data, we propose the use approximately-secure and approximately-correct MAC (AMAC) introduced by Di Crescenzo et al. [31]. The system verifies authenticity while tolerating a small amount of modification, loss, or corruption of the original data. The exact level of tolerance can be tuned. A more detailed construct can be found in Appendix A.

We parallelize the AMAC construction to adapt it to file systems; in addition, we propose to use PMAC as building block in the AMAC construction [31], to allow for incremental update. The atom for the computation is a file system block, rather than a bit. The approximate security and correctness then refer to the number of corrupted or missing blocks, rather than bits. We give details of the algorithms for AMAC using PMAC in the Appendix but we leave a formal treatment of incremental AMACs for future work.

The chief benefit of using the AMAC construction over regular MAC constructions lies in verification. Serial and parallel MACs require the entire message as input to verify authenticity. Using AMAC, a portion of the original message can be ignored. This allows a weaker statement of authenticity to be constructed even when some data are unavailable. The drawback of AMAC lies in the reduction of authenticity. With AMAC, some data may be acceptably modified in the original source.

Chapter 5

Conclusions

"We're just gettin' started!"

– Ace Frehley

Health care providers, private companies, and federal agencies are now subject to sweeping regulations that affect the management of their electronic records. These include: the Health Insurance Portability and Accountability Act (HIPAA), the Gramm-Leach-Bliley Act (GLBA), the Federal Information Security Management Act (FISMA) and the Sarbanes-Oxley Act (SOX). However, this legislation is unclear about which technologies companies are to implement in order to be compliant. We distilled federal electronic record management legislation into four technological categories: versioning with real-time access, secure deletion, digital privacy and digital authenticity. We address each of these categories with a technological solution designed to meet the requirements of electronic record legislation.

5.1 Summary of Contributions

We have implemented three contributions to the field of regulatory compliant storage. The first, ext3cow, is a fully implemented open-source file system that provides users with a new and intuitive interface for accessing data in the past. Ext3cow's versioning interface supports many features: file system snapshot, per-file versioning, version enumeration, and a continuous-time view of changes to a file system. To provide these functions, ext3cow uses a copy-on-write scheme and versioning metadata that incur little overhead and exhibit a small data footprint. All modifications made to ext3cow are encapsulated within the on-disk file system, avoiding the disadvantages of kernel (virtual file system) or user-space implementations. Given these features, ext3cow 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. However, ext3cow was specifically designed for the management of data in compliance with federal electronic records legislation. As it stands, ext3cow meets the mandated versioning and auditability requirements. In addition, ext3cow's file organization is suitable for our implementation of secure deletion.

To this end, we defined a model for secure deletion in storage systems that share data between files, specifically, versioning file systems that comply with federal regulations. Our model supports authenticated encryption, a unique feature for file systems. A data block is encrypted and converted into a ciphertext block and a small stub. Securely overwriting the stub makes the corresponding block irrecoverable.

We present two algorithms within this model. The first algorithm employs the all-or-nothing transform so that securely overwriting a stub or any 128 bits of a ciphertext securely deletes

the corresponding block. The second algorithm generates a random key per block in order to make encryption non-repeatable. The first algorithm produces more compact stubs and supports a richer set of deletion primitives. Whereas the second algorithm provides stronger privacy guarantees.

Both secure deletion algorithms meet our requirement of minimizing secure overwriting, resulting in a 200 times speed-up over previous techniques. The addition of stub metadata and a cryptographic device driver degrade performance minimally. We have implemented secure deletion in the ext3cow versioning file system for Linux and in a secure device driver.

Lastly, we have introduced a model for digital audits of versioning file systems that supports compliance with federally mandated data retention guidelines. In this model, a file system commits to a version history by transmitting audit metadata to a third party. This prevents the owner of the file system (or a malicious party) from modifying past data without detection. Our techniques for the generation of audit metadata use incremental authentication methods that are efficient when data modifications are fine grained, as in versioning file systems. Additionally, authentication methods are resilient to data loss or temporary outages. Experimental results show that incremental authentication can perform up to 94% faster than traditional sequential authentication algorithms. We have implemented incremental authentication in ext3cow and, like all technologies presented, available at: *www.ext3cow.com*.

Appendix A

The AMAC Construct

The AMAC Construct (see [31]): Let M denote the message space where $m \in M$ is an instance of a message, let d represent a distance function computed over M (such as the hamming distance), and let k represent a secret key. An *approximately-secure and approximately-correct MAC for distance function d* is represented by an authentication tag generation algorithm **Tag**(m,k,d) that computes the AMAC and returns the value *tag*, and a verification algorithm **Verify**(m,k,tag,d) that returns **true** if and only if $tag =$**Tag**(m,k,d).

An AMAC has $(d,p,\delta)-$*approximate-correctness* if *tag*=**Tag**(m,k,d), then with probability at least p **Verify**(m',k,tag,d) will return **true** if $d(m,m') \leq \delta$. An AMAC has $(d,\gamma,t,q,\varepsilon)-$*approximate-security* if an adversary operating in time t makes q queries to a tag generation oracle, the probability that the adversary can construct a message m' such that $d(m,m') \geq \gamma$ and **Verify**(m',k,tag,d) returns **true**, is at most ε.

Tag and Verify (see [31]): To construct an AMAC tag using the **Tag** algorithm, perform the following steps. Each AMAC also takes as input a counter *ct* that seeds randomness in the **Tag** and **Verify**

algorithms; *ct* should not be reused.

1. Set $x_1 = \lceil n/2c\delta \rceil$, where n is the size of the message in bits and c is a pre-specified block size in bits.

2. Set $x_2 = \lceil 10log(1/(1-p)) \rceil$.

3. Write $\pi(m \oplus L)$ as $m_1|m_2|\ldots|m_{\lceil n/c \rceil}$, where L is a random bit string and π is a random permutation both unique given the value of *ct*, and each m_i represents a block of size c of the manipulated message.

4. Using randomness based on *ct*, create x_2 message subsets, $S_1, S_2, \ldots, S_{x_2}$, where each subset is the concatenation of x_1 randomly chosen blocks m_i.

5. For each subset, compute $sh_i = H(S_i, k)$, where H can be implemented as a secure MAC (formally it has to be a target collision resistant function) and k is retrieved from randomness based on the seed *ct*.

6. Return as the final tag, $ct|sh_1|sh_2|\ldots|sh_{x_2}$.

The **Verify** algorithm performs steps 1 through 5 of the above algorithm on message m' acquiring sub-tags $sh'_1, sh'_2, \ldots, sh'_{x_2}$. **Verify** then returns **true** if and only if $sh_i = sh'_i$ for at least αx_2 sub tags, where $\alpha = 1 - 1/2\sqrt{e} - 1/2e$.

Constructing and verifying tags allows for the original input to be partially modified, corrupted or even missing for up to δ bits, and still maintain approximate correctness and security so long as the underlying function H is a universal one-way hash function [82].

Update (Incremental AMAC): An AMAC based on a parallel MAC can be efficiently updated when only a portion of the original message has changed; only the modified block is needed.

Our idea is to replace H with a parallel MAC. We are assuming that it is possible to build a (finite) family of universal one-way hash functions from the PMAC construction (or from other deterministic parallel MAC constructions).

The **Update** algorithm takes as input an original message block b, a modified message block b', the position of the modified block within the original input data source r, and the authenticator tag being updated $ct|sh_1|\ldots|sh_{x_2}$.

1. Set $x_1 = \lceil n/2c\delta \rceil$, where n is the size of the message in bits and c is a pre-specified block size in bits.
2. Set $x_2 = \lceil 10log(1/(1-p)) \rceil$.
3. Use $\pi(m)$ to compute the permuted position of the modified block in the message.
4. Using randomness based on ct, determine the subsets and positions within each subset where block b is used.
5. Since we are using PMAC, we can efficiently update each sub tag after computing each subset and position where b is placed. Compute $sh'_i = sh_i \oplus Y(b) \oplus Y(b')$, where $Y(\cdot)$ is computed as in Section 4.4.1.
6. Return as the updated tag, $ct|sh'_1|sh'_2|\ldots|sh'_{x_2}$.

Initially computing the authenticator value for a portion of the file system using AMAC requires the entire input source to be accessed, just as it would if using a conventional PMAC algorithm. Computationally there is more work to be performed when computing each AMAC, but memory operations are negligible when compared with disk I/O. Updating an incremental AMAC requires the same number of disk accesses as updating a PMAC.

Bibliography

[1] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato. IETF RFC 3161 time-stamp protocol (tsp). IETF Network Working Group, 2001.

[2] R. Anderson. The dancing bear – a new way of composing ciphers. In *Proceedings of the International Workshop on Security Protocols*, April 2004.

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

[4] 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.

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

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

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

[8] M. Bellare and C. Namprempre. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt'00 Proceedings*, volume 1976. Springer-Verlag, 2000. Lecture Notes in Computer Science.

[9] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - Eurocrypt'02 Proceedings*, volume 2332, pages 384 – 397. Springer-Verlag, 2002. Lecture Notes in Computer Science.

[10] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the ACM conference on Computer and Communications Security*, pages 9–16, November 1993.

[11] M. Blaze. High-bandwidth encryption with low-bandwidth smartcards. In *Fast Software Encryption*, volume 1039, pages 33–40, 1996. Lecture Notes in Computer Science.

[12] M. Blaze, J. Feigenbaum, and M. Naor. A formal treatment of remotely keyed encryption. In *Advances in Cryptology – Eurocrypt'98 Proceedings*, volume 1403, pages 251–265, 1998. Lecture Notes in Computer Science.

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

[14] V. Boyko. On the security properties of OAEP as an all-or-nothing transform. In *Advances in Cryptology - Crypto'99 Proceedings*, pages 503–518. Springer-Verlag, August 1999. Lecture Notes in Computer Science.

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

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

[17] 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.

[18] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium*, pages 331–346, August 2005.

[19] 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 Winter USENIX Technical Conference*, pages 43–60, 1992.

[20] U.S. Securities Exchange Commission. Commission guidance to broker-dealers on the use of electronic storage media under the electronic signatures in global and national commerce act of 2000 with respect to rule 17a-4(f). SEC Release No. 34-44238, May 2001.

[21] United States Congress. The Sarbanes-Oxley Act. 17 C.F.R. Parts 228, 229 and 249.

[22] United States Congress. The Health Insurance Portability and Accountability Act, 1996.

[23] United States Congress. The Health Insurance Portability and Accountability Act Privacy Rule. 67 FR 53182, 1996.

[24] United States Congress. The Gramm-Leach-Bliley Act. 15 USC, Subchapter I, § 6801-6809, 1999.

[25] United States Congress. Federal Information Security Management Act. Public Law 107-347, USC 44 Chapter 35, Subchapter III Information Security, 2002.

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

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

[28] Symantec Corporation. Understanding and complying with FISMA. www.symantec.com, February 2004.

[29] Symantec Corporation. SOX compliance: Understanding how security, systems and storage management solutions help meet the corporate demand for SOX compliance. www.symantec.com, January 2005.

[30] 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*, volume 1563, pages 500–509. Springer-Verlag, 1999. Lecture Notes in Computer Science.

[31] G. Di Crescenzo, R. Graveman, R. Ge, and G. Arce. Approximate message authentication

and biometric entity authentication. In *Proceedings of Financial Cryptography and Data Security*, February-March 2005.

[32] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–215, October 2001.

[33] DataMirror. Achieving Sarbanes-Oxley compliance with real-time data integration, protection and monitoring. www.datamirror.com, September 2003.

[34] Deloitte and LLP Touche. Leveraging internal control to build a better business: A response to Sarbanes-Oxley sections 302 and 404. www.deloitte.com, April 2003.

[35] Y. Dodis and J. An. Concealment and its applications to authenticated encryption. In *Advances in Cryptology – Eurocrypt'03 Proceedings*, volume 2656, 2003. Lecture Notes in Computer Science.

[36] R. Dowdeswell and J. Ioannidis. The CryptoGraphic disk driver. In *Proceedings of the USENIX Technical Conference, FREENIX Track*, pages 179–186, June 2003.

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

[38] P. Cederqvist *et. al*. *Version Management with CVS*. Network Theory Limited, 2003. http://www.network-theory.co.uk/cvs/manual/.

[39] D. Farmer and W. Venema. *Forensic Discovery*. Addison-Wesley, 2004.

[40] K. Fu, M. F. Kasshoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, 2002.

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

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

[43] 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.

[44] D. Grune, B. Berliner, and J. Polk. Concurrent versioning system (CVS). http://www.cvshome.org/, 2003.

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

[46] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the USENIX Security Symposium*, pages 243–257, January 1998.

[47] P. Gutmann. Data remanence in semiconductor devices. In *Proceedings of the USENIX Security Symposium*, pages 39–54, August 2001.

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

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

[50] E. Haubert, J. Tucek, L. Brumbaugh, and W. Yurcik. Tamper-resistant storage techniques for multimedia systems. In *IS&T/SPIE Symposium Electronic Imaging Storage and Retrieval Methods and Applications for Multimedia (EI121)*, pages 30–40, January 2005.

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

[52] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter USENIX Technical Conference*, pages 235–246, January 1994.

[53] 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.

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

[55] Kahn Consulting Inc. The Sarbanes-Oxley Act: Understanding the implications for information and records management. www.KahnConsultingInc.com.

[56] Kahn Consulting Inc. An evaluation of the Sun Microsystems, Inc. *STOREEDGE* compliance archiving system. www.KahnConsultingInc.com, January 2005.

[57] M. Jakobsson, J. Stern, and M. Yung. Scramble all. Encrypt small. In *Fast Software Encryption*, volume 1636, 1999. Lecture Notes in Computer Science.

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

[59] 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.

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

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

[62] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummandi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the ACM Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, November 2000.

[63] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.

[64] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, April 1988.

[65] J .P. Lucci. Enron–the bankruptcy heard around the world and the international ricochet of Sarbanes-Oxley. 67 Alb. L. Rev. 211, 2003.

[66] J. P. MacDonald, P. N. Hilfinger, and L. Semenzato. PRCS: The project revision control system. In *Proceedings of System Configuration Management*, volume 1439. Springer-Verlag, July 1998. Lecture Notes in Computer Science.

[67] J. R. Macey. Pox on both your houses: Enron, Sarbanes-Oxley and the debate concerning the relative efficacy of mandatory versus enabling rules. 81 Wash. U. L.Q. 329, 333, 2003.

[68] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 31–46, January 2002.

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

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

[71] 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.

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

[73] 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.

[74] S. Micali. Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, Massachusetts Institute of Technology, 1996.

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

[76] J. Monroe. Emerging solutions for content storage. Presentation at PlanetStorage, 2004.

[77] J. Morris. The Linux kernel cryptographic API. *Linux Journal*, (108), April 2003.

[78] 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.

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

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

[81] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187, October 2001.

[82] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 33–43, May 1989.

[83] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A

system for migrating computing environments. In *Proceedings of the USENIX ymposium on Operating Systems Design and Implementation (OSDI)*, pages 361–376, 2002.

[84] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok. I^{3}FS: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the Large Installation System Administration Conference (LISA)*, pages 67–78, November 2004.

[85] 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 USENIX Conference on File and Storage Technologies (FAST)*, pages 117–129, January 2002.

[86] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[87] Z. N. J. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin. Secure deletion for a versioning file system. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)*, pages 143–154, December 2005.

[88] Z. N. J. Peterson, R. Burns, and A. Stubblefield. Limiting liability in a federally compliant file system. In *Proceedings of the PORTIA Workshop on Sensitive Data in Medical, Financial, and Content Distribution Systems*, July 2004.

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

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

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

[92] S. Ranade. The time traveling file manager: Interface design and semantics. Technical report, The Johns Hopkins University, 2005.

[93] KCI Research. New information management rules needed for audit, investigations, and litigation. www.KahnConsultingInc.com, September 2004.

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

[95] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.

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

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

[98] 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.

[99] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation

of the Sun network file system. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–130, June 1985.

[100] D. J. Santry, M. J. Feeley, N. C Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999.

[101] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

[102] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems Security*, 2(2):159–176, 1999.

[103] M. Scholl, R. Kissel, S. Skolochenko, and X. Li. Guidelines for media sanitization. NIST Special Publication 800-88, February 2006.

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

[105] 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 USENIX Technical Conference*.

[106] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[107] J. S. Shapiro and J. Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *Proceedings of the USENIX Technical Conference, FREENIX Track*, pages 203–216, 2002.

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

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

[110] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

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

[112] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, October 2000.

[113] A. Stubblefield, J. Ioannidis, and A. D. Rubin. Using the Fluhrer, Mantin, and Shamir attack to break WEP. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 17–22, February 2002.

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

[115] W. F. Tichy. RCS: A system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

[116] T. Y. Ts'o and S. Tweedie. Planned extensions to the Linux ext2/ext3 filesystem. In *Proceedings of the USENIX Technical Conference, FREENIX Track*, pages 235–243, June 2002.

[117] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[118] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, Web publishing system. In *Proceedings of the USENIX Security Symposium*, pages 59–72, August 2000.

[119] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology - Crypto'05 Proceedings*. Springer-Verlag, August 2005. Lecture Notes in Computer Science. To appear.

[120] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proceedings of the Workshop on Future Directions in Distributed Computing*, pages 142–147, June 2002.

[121] C. Wright, J. Dave, and E. Zadok. Cryptographic file systems performance: What you don't know can hurt you. In *Proceedings of the IEEE Security in Storage Workshop (SISW)*, pages 47–61, October 2003.

[122] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the USENIX Technical Conference*, pages 197–210, June 2003.

[123] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, May 1999.

[124] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the USENIX Technical Conference*, pages 55–70, June 2000.

[125] J-G. Zhu, Y. Luo, and J. Ding. Magnetic force microscopy study of edge overwrite characteristics in thin film media. *IEEE Transaction on Magnetics*, 30(6):4242–4244, 1994.

Vita

Zachary Nathaniel Joseph Peterson was born February 11, 1978 in Akron, Ohio. At the age of four, his family moved to Escondido, California, where he completed his basic and high school education. In 1996, he began attending the University of California at Santa Cruz. He graduated four years later with a Bachelor of Science in Computer Engineering with liberal arts emphasis in music. He stayed on at Santa Cruz to complete a Master of Science in Computer Science under the guidance of Prof. Darrell Long. The title of his Master's thesis was *Data Placement for Copy-on-Write Using Virtual Contiguity*. In 2002, Zachary matriculated at The Johns Hopkins University. While there, he earned a Masters of Science in Security Informatics from the Johns Hopkins Institute for Security Informatics under the advisement of Avi Rubin. He successfully defended his dissertation in October of 2006, completing the requirements for a Doctor of Philosophy in Computer Science. His adviser was Randal Burns.