Design and Implementation of Verifiable Audit Trails for a Versioning File System[†]

Zachary N. J. Peterson
Johns Hopkins University

Randal Burns
Johns Hopkins University

Giuseppe Ateniese
Johns Hopkins University

Stephen Bono
Johns Hopkins University

Abstract

We present 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. Digital audit trails create an analog of the paper audit process for file data, helping to meet the requirements of electronic records legislation. Our techniques address the I/O and computational efficiency of generating and verifying audit trails, the aggregation of audit information in directory hierarchies, and independence to file system architectures.

1 Introduction

The advent of Sarbanes-Oxley (SOX) [40] 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 the 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 [14, 22, 33, 27, 38] and provenance-aware storage systems [26] 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 records through time-oriented file system interfaces [31]. 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 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 of the data itself, nor does it prevent data destruction. 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 a 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 privacy-preserving, one-way functions of file data to third parties. Published data may even be stored publicly, *e.g.* on a Web page.

[†]This is the fully developed version of a work-in-progress paper that appeared as a *short paper* at the 2005 ACM StorageSS Workshop [7].

Our 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 [13]. We employ incremental message authentication codes [4, 5, 6] that allow MACs to be computed based only on data that have changed from the previous version. Incremental MAC 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. Our protocol also reduces network 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.

Our solution is based on keyed, cryptographic hash functions, such as HMAC-SHA1 [3]. Public-key methods for authenticating data exist [28] and provide unique advantages over symmetric-key solutions. For instance, during an audit, a file system would reveal its public key to the auditor, allowing the auditor to verify data authenticity only. The auditor would not have the ability to create new, authentic records. With symmetric-key hash functions, when the key is revealed to the auditor, the auditor could also create authentic records, leaving open the possibility of falsifying data. This is out of the scope of our attack model. The auditor is a trusted and independent entity. In this paper, we do not consider a public-key implementation, because public-key operations are far too costly to be used in practice.

Our techniques are largely file system independent in that they do not require a specific metadata architecture. This allows verifiable audit trails to be implemented on a wide variety of systems. Additionally, our design makes the audit robust to disk failures, immune to backup and restore techniques, and allows for easy integration into information life-cycle management (ILM) systems.

We have implemented authentication using incremental MACs in the ext3cow file system. Ext3cow is a freely-available, open-source file system designed for version management in the regulatory environment [31]. Experimental results show that incremental MACs increase performance by 94% under common workloads when compared with traditional, serial hash MACs.

2 Related Work

Most closely related to this work is the SFS-RO system [13], 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 an erasure-coded, distributed block. This detects corruption without relying on error correction and provides authenticity [42]. Patil *et al.* [30] 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.* [15] 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 [37]. 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.

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.

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, 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 [41].

The principal attack against which our 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; inserting or deleting versions into a sequence without detection.

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. Disk drives may be accessed directly through the device interface and on-disk structures are easily examined and modified [12]. 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 [39].

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 are alone 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 [24]. 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, which helps prevent audit failures after committing to a version history.

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

$$A_{v_i} = \text{MAC}_K(v_i||A_{v_{i-1}}); A_{v_0} = \text{MAC}_K(v_0||N) \quad (1)$$

in which K is an authentication key and N is a nonce. N is a randomly derived value that differentiates the authenticators for files that contain the same data, including empty files. We also require that the MAC function reveals nothing about the content of the data. Typical MAC constructions provide this property. CBC-MAC [2, 16] and HMAC-SHA1 [3] suffice.

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 and couples past versions to the current authenticator. The wide application of one-way hash chains in password authentication [20], micropayments [34], and certificate revocation [23] testifies to their utility and security.

The authentication key K binds each MAC to a specific identity and audit scope. K is a secret key that is selected by the auditor. This ensures keys are properly formed and meet the security requirements of the system. During an audit, the auditor verifies all version histories authenticated with K. Keys may be generated to bind a version history to an 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 [17], 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 [21]. 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 a single version, the audi-

tor requests version data v_i and the previous version authenticator $A_{v_{i-1}}$ from the file system, computes A_{v_i} using Equation 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 $A_{v_{i-1}}$ has not been verified by this audit.

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 are identical to the data used to generate the version identifiers, *including all intermediate versions*.

Verifying individual versions and version histories relies upon the collision resistant properties of MACs. For individual versions, the auditor uses the unverified and untrusted $A_{v_{i-1}}$ from the file system. A_{v_i} authenticates version v_i even when an adversary can choose input $A_{v_{i-1}}$. Finding a replacement for $A_{v_{i-1}}$ and v_i that produces the correct A_{v_i}, finds a hash collision. 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 finds 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. We quantify this trade-off in Section 6.3.

4.1 Incrementally Calculable MACs

I/O efficiency is the principal concern in the calculation and verification of version authenticators in a file system. A version of a file shares data with its predecessor. It differs only in the blocks of data that are changed. As a consequence, the file system performs I/O only on these changed blocks. For performance reasons, it is imperative that the system updates audit trails based only on the changed data. To achieve this, we rely on incremental MAC constructions that allow the MAC of a new version to be calculated using only the previous MAC and the data that have changed. Thus, MAC computation performance scales with the amount of data that are written, rather than size of the file being MACed.

Typical MAC constructions, such as HMAC [3] and CBC-MAC [2, 16], are serial in nature; they require the entire input data to compute the MAC. HMAC relies on a standard hash function H, such as SHA1 [29], which is called twice as

$$H_K(M) = H(K \oplus \text{pad1} || H(K \oplus \text{pad2} || M)).$$

HMAC is very efficient. It costs little more than a single call of the underlying hash function – the outer hash is computed on a very short input. However, HMAC is serial because all data are used as input to the inner hash function. CBC-MAC builds on a symmetric cipher used in CBC mode. In particular, given a message M, divided in blocks $M_1, \ldots M_k$, and a cipher $E_K(\cdot)$, it computes $O_1 = E_K(M_1)$ and $O_i = E_K(M_i \oplus O_{i-1})$, for $2 \leq i \leq k$. CBC-MAC(M) is then the final value O_k. CBC-MAC is inherently serial because the computation of O_i depends on the previous value O_{i-1}.

We use the XOR MAC construction [5], which improves on CBC-MAC, making it incremental and parallelizable. XOR MAC (XMACR in Bellare [5]) builds upon a block cipher $E_K(\cdot)$ in which the block size is n. A message M is divided into blocks, each of a certain length m, as $M = M_1 \ldots M_k$ (M_k is padded if its length is less than m). Then XOR MAC(M) is computed as (r, Z), for a random seed r, and

$$Z = E_K(0||r) \oplus \left[\bigoplus_{j=1}^{k} E_K(1||\langle j \rangle||M_j) \right] \qquad (2)$$

in which $0, 1$ are bits and $\langle j \rangle$ is the binary representation of block index j. The leading bit differentiates the contribution of the random seed from all block inputs. The inclusion of the block index prevents reordering attacks. Reordering the message blocks results in different authenticators. When using AES-128 [10] for $E_K(\cdot)$, $n = 128$ and $|r| = 127$ bits. When using 47 bits for the block index $\langle j \rangle$, XOR MAC makes an AES call for every 80 bits of the message M.

Our implementation of XOR MAC aligns the block sizes used in the algorithm to that of file system blocks: $|M_j| = 4096$ bytes. As suggested by the original publication [5], a keyed hash function can be used in place of a block cipher to improve performance. We use HMAC-SHA1 to instantiate E_K.

XOR MAC provides several advantages when compared with CBC-MAC or HMAC. It is parallelizable in that the calls to the block cipher can be made in parallel. This is important in high-speed networks, multiprocessor machines, or when out-of-order verification is needed [5], for instance when packets arrive out-of-order owing to loss and retransmission. Most important

to our usage, XOR MAC is incremental with respect to block replacement. When message block M_j has been modified into M'_j, it is possible to compute a new MAC (r', Z'), for a fresh random value r', on the entire M by starting from the old value (r, Z) as

$$T = Z \oplus E_K(0||r) \oplus E_K(1||\langle j \rangle||M_j)$$
$$Z' = T \oplus E_K(0||r') \oplus E_K(1||\langle j \rangle||M'_j)$$

XORing out the contributions of the old block and old random seed to make T and XORing in the contributions of the new block and new random seed to build Z'. File systems perform only block replacements. They do not insert or delete data, which would change the alignment of the blocks within a file.

PMAC [6] improves upon XOR MAC in that it makes fewer calls to the underlying block cipher. XOR MAC expands data by concatenating an index to the message block. PMAC avoids this expansion by defining a sequence of distinct *offsets* that are XORed with each message block. Thus, it operates on less data, resulting in fewer calls to the underlying block cipher. Indeed, we initially proposed to use PMAC in our system [7].

However, when XOR MAC or PMAC are instantiated with keyed hash functions (rather than block ciphers), the performance benefits of PMAC are minimal for file systems. The reason is that HMAC-SHA1 accepts large inputs, permitting the use of a 4096 byte file system block. The incremental cost of a 64 bit expansion, representing a block index, is irrelevant when amortized over a 4096 byte block. At the same time, XOR MAC is simpler than PMAC and easier to implement. (On the other hand, PMAC is deterministic, requires no random inputs, and produces smaller output). In our system, we elect to implement XOR MAC.

4.2 XOR MAC for Audit Trails

We use the incremental property of XOR MAC to perform block-incremental computation for copy-on-write file versions. Each version v_i comprises blocks $b_{v_i}(1), \ldots, b_{v_i}(n)$ equal to the file system block size and a file system independent representation of the version's metadata, denoted $\overline{M}_{v_i}$ (see Section 4.3). The output of XOR MAC is the exclusive-or of the one-way functions

$$A_{v_i} = H_K(00||r_{v_i}) \oplus \left[\bigoplus_{j=1}^{n} H_K(01||\langle j \rangle||b_{v_i}(j))\right] \oplus H_K(10||\overline{M}_{v_i}) \oplus H_K(11||A_{v_{i-1}}) \quad (3)$$

in which r_{v_i} is a random number unique to version v_i. This adapts equation 2 to our file system data. We have added an additional leading bit that allows for four distinct components to the input. Bit sequences 00, 01, and 10 precede the random seed, block inputs, and normalized metadata respectively. To these, we add the previous version authenticator, which forms the version hash chain defined by equation 1. This form is the full computation and is stored as the pair (r, A_{v_i}). There is also an incremental computation. Assuming that version v_i differs from v_{i-1} in one block only $b_{v_i}(j) \neq b_{v_{i-1}}(j)$, we observe that

$$\begin{aligned} A_{v_i} =& A_{v_{i-1}} \oplus H_K(00||r_{v_i}) \oplus H_K(00||r_{v_{i-1}}) \\ & \oplus H_K(01||\langle j \rangle||b_{v_i}(j)) \oplus H_K(01||\langle j \rangle||b_{v_{i-1}}(j)) \\ & \oplus H_K(10||\overline{M}_{v_i}) \oplus H_K(10||\overline{M}_{v_{i-1}}) \\ & \oplus H_K(11||A_{v_{i-1}}) \oplus H_K(11||A_{v_{i-2}}). \end{aligned}$$

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 contributions of the past version authenticator, normalized metadata, and random seed.

The computation of XOR MAC authenticators scales with the amount of I/O, whereas the performance of a hash message authentication code (HMAC) scales with the file size. With XOR MAC, only new data being written to a version will be authenticated. HMACs must process the entire file, irrespective of the amount of I/O. This is problematic as studies of versioning file systems show that data change at a fine granularity [31, 38]. Our results (Section 6) confirm the same. More importantly, the computation of the XOR MAC version authenticator requires only those data blocks being modified, which are already in cache, requiring little to no additional disk I/O. Computing an HMAC may require additional I/O. This is because system caches are managed on a page basis, leaving unmodified and unread 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.

4.3 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 and remains valid when the physical implementation of a file changes. This includes transfers of a file from system to system (with the caveat that all systems storing data support audit trails – we have implemented only one). The act of performing a data restoration may be a procedure worth auditing in and of itself. We consider this outside the scope of the file system requirements.

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: name, 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 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 3), a directory authenticator would be the MAC of its data (directory entries), the MAC of the previous directory authenticator and its normalized inode information. However, this construct fails to bind the data of a directory's files to the names, allowing an attacker to undetectably exchange the names of files within a directory.

We employ trees of MACs that bind individual versions and their names to a file system hierarchy, authenticating the entire versioning file system recursively. In addition to the normalized inode information and previous authenticator used to authenticate files, directory authenticators are composed of name-authenticator pairs. For each file within the directory, we concatenate its authenticator to the corresponding name and take a one-way hash of the result.

$$A_{D_i} = H_K(00||r_{D_i}) \oplus \left[\bigoplus_{j=1}^{n} H_K(01||\langle j\rangle||name_j||A_{v_j})\right] \oplus H_K(10||\overline{M}_{D_i}) \oplus H_K(11||A_{D_{i-1}})$$

This binds files and sub-directories to the authenticator of the parent directory. Directory version authenticators continue recursively to the file system root, protecting the entire file system image. The SFS-RO system [13] used a similar technique to fix the contents of a read-only file system without versioning. Our method differs in that it is incremental and accounts for updates.

For efficiency reasons, we bind versions to the directory's authenticator lazily. Figure 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 directory contents change; *i.e.*, files are created, destroyed, or renamed. In this example, when deleting file U (Figure 1), the authenticator is updated to the current versions. Alternatively, 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 as it would need to be done on every write.

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. A rename moves a file from one directory's scope to another. 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 (see Section 4). When auditing a directory at a given point

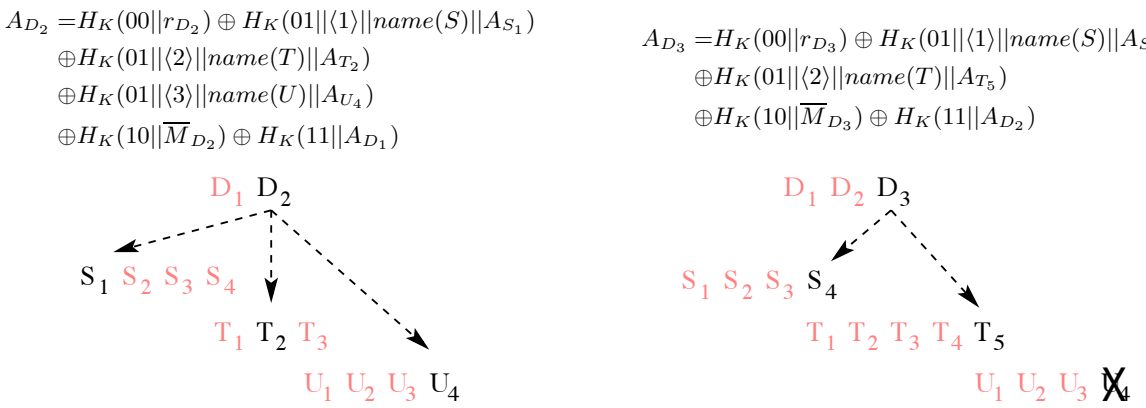


Figure 1: Directory version authenticators before and after file U is deleted. Equations show the full (not incremental) computation.

in time, the auditor must access the directory at the time when its was created and then follow the children files' hash chains forward to the specified point in time. Updating directory authenticators more frequently may be desirable to speed the audit process.

5 File System Implementation

We have implemented digital audit trails using XOR MAC in ext3cow [31], an open-source, block-versioning file system designed to meet the requirements of electronic records legislation. Ext3cow supports file system snapshot, per-file versioning, and a time-oriented interface. 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. We store version authenticators for a file in its inode. We have already retrofitted the metadata structures of ext3cow to support versioning and secure deletion (based on authenticated encryption [32]). Version authenticators are a straightforward extension to ext3cow's already augmented metadata, requiring only a few bytes per inode.

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 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 ($H_K(11||A_{v_{i-1}})$), 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 ($H_K(11||A_{v_{i-2}})$). Each authenticator computation requires access to the previous and penult-previous authenticators and their hashes. By storing authenticators and hashes for previous versions together, the system avoids 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 ($H_K(11||A_{v_{i-1}})$) to the penult-previous authenticator ($A_{v_{i-2}}$) and hash ($H_K(11||A_{v_{i-2}})$). The once current authenticator (A_{v_i}) is zeroed, and is calculated on an as-needed basis.

In almost all cases, authenticator blocks do not increase the number of disk seeks performed by the system. The block allocator in ext3cow makes efforts to collocate data, metadata, and authenticator blocks in a single disk drive track, maintaining contiguity. Authenticator blocks are very likely to be read out of the disk's track cache. The same disk movement that reads inode or data blocks populates the track cache.

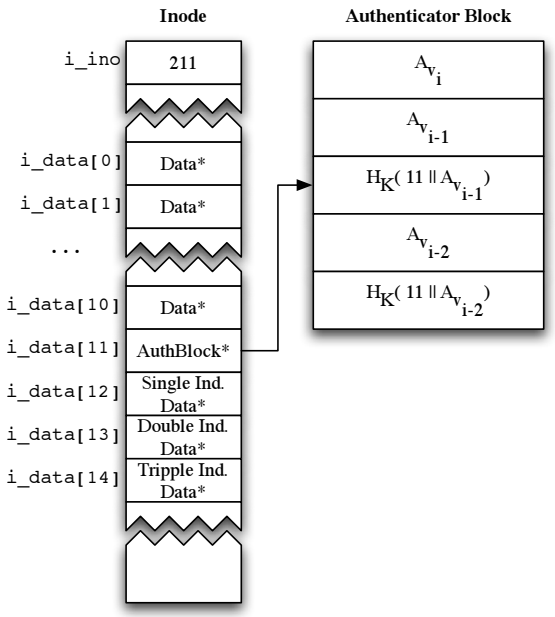


Figure 2: Metadata architecture to support version authenticators.

5.2 Key Management

Key management in ext3cow uses lockboxes [17] 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 [32].

6 Experimental Results

We measure the impact of authentication on versioning file systems and compare the performance characteristics of HMAC and XOR MAC in the ext3cow versioning file system. We begin by comparing the CPU and disk throughput performance of HMAC and XOR MAC 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 XOR MAC. 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, XOR MAC and HMAC, were implemented in the ext3cow file system using the standard HMAC-SHA1 keyed-hash function provided by the Linux kernel cryptographic API [25]. For brevity, XOR MAC implemented with HMAC-SHA1 is further referred to as XOR MAC-SHA1. 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.

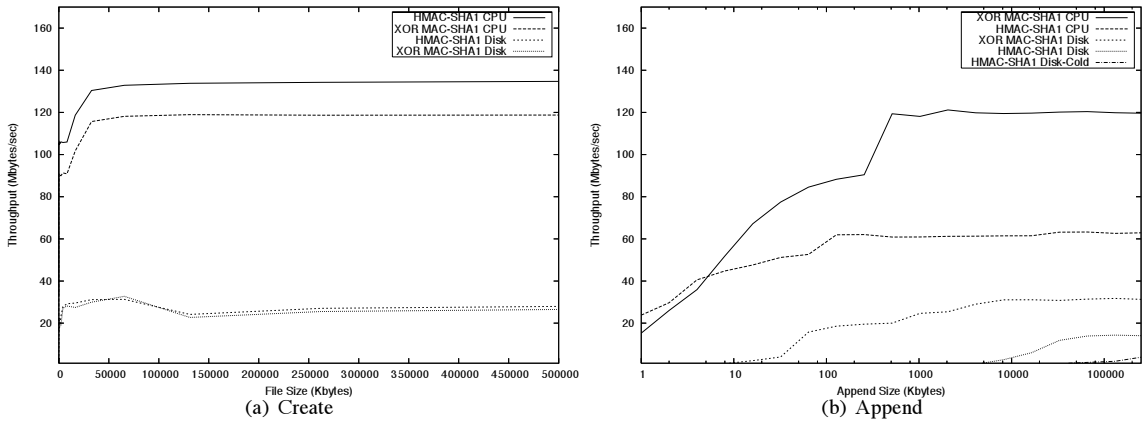
6.1 Micro-benchmarks

To quantify the efficiency of XOR MAC, 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 XOR MAC, an append requires only a MAC of a new random value, 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 data 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 3(a) presents the results of the *create* micro-benchmark. Traditional HMAC-SHA1 has higher CPU throughput than XOR MAC-SHA1, saturating the CPU at 134.8 MB/s. The XOR MAC achieves 118.7 MB/s at saturation. This is expected as XOR MAC-SHA1 performs two calls to SHA1 for each block (see Equation 3), compared to HMAC-SHA1 that only calls SHA1 twice for each file, resulting in additional computation time. Additionally, SHA1 appends the length of the message that it's hashing to the end of the message, padding up to 512-bit boundaries. Therefore, XOR MAC-SHA1 hashes more data, up to n*512 bits more for n blocks.

Despite XOR MAC's computational handicap, disk throughput measurements show little performance disparity. HMAC-SHA1 achieves a maximum of 28.1 MB/s and XOR MAC-SHA1 a maximum of 26.6 MB/s. This illustrates that calculating new authenticators for a file system is I/O-bound, making XOR MAC-SHA1's ultimate performance comparable to that of HMAC-SHA1.

The results of the *append* micro-benchmark make a compelling performance argument for incremental MAC



(a) Create (b) Append

Figure 3: Results of micro-benchmarks measuring the CPU and disk throughput.

computation. Figure 3(b) shows these results – note the log scale. We observe XOR MAC-SHA1 outperforms HMAC-SHA1 in both CPU and disk throughput measurements. XOR MAC-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, XOR MAC-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. These performance gains arise from the incremental nature of XOR MACs. 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 XOR MAC.

6.2 Aggregate Performance

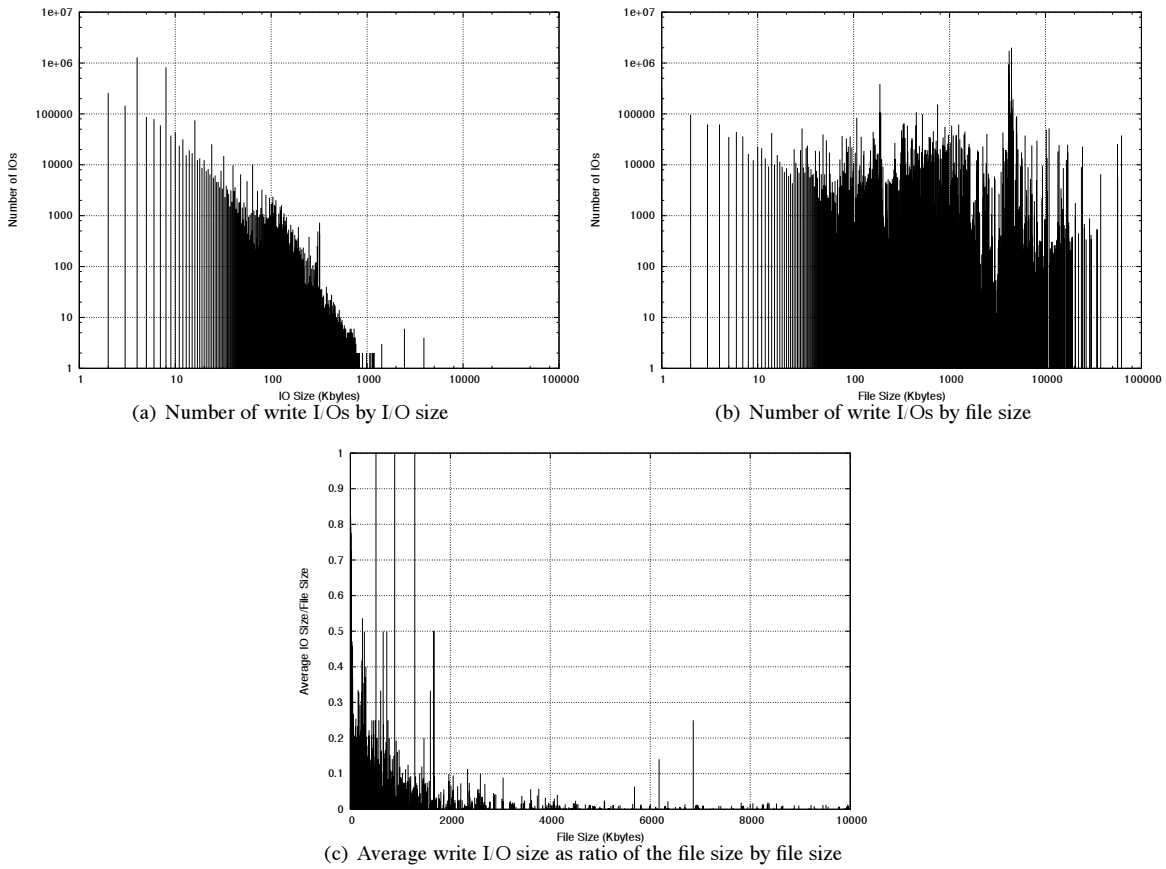
We take a broader view of performance by quantifying the aggregate benefits of XOR MAC on a versioning file system. To accomplish this, we replayed four months of system call traces [35] on an 80 gigabyte ext3cow partition, resulting in 4.2 gigabytes of data in 81,674 files. Despite their age, these 1996 traces are the most suitable for these measurements. They include information that allow multiple open/close sessions on the same file to be correlated – necessary information to identify versioning. More recent traces [11, 19, 36, 38] do not include adequate information to correlate open/close sessions, are taken at too low a level in the IO system to be useful, or would introduce new ambiguities, such as the effects of a network file system, into the aggregate measurements. Our experiments compare trace-driven throughput performance as well as the total computation costs for performing a digital audit using the XOR MAC 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.

No Authentication	XOR MAC	HMAC
1.98 MB/s	1.77 MB/s	0.11 MB/s

Table 1: The trace-driven throughput of no authentication, XOR MAC and HMAC.

6.2.1 Write Performance

The incremental computation of XOR MAC minimally degrades on-line system performance when compared with a system that does not generate audit trails (No Authentication). In contrast, HMAC audit trails reduce throughput by more than an order of magnitude (Table 1). 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 XOR MAC-SHA1 authentication. XOR MAC-SHA1 achieves a 93.9% improvement in run-time performance over HMAC-SHA1: 1.77 MB/s versus 0.11 MB/s. HMAC-SHA1's degradation results from the additional read I/O and computation time it must perform on every write. XOR MAC-SHA1 incurs minimal performance penalties owing to its ability to compute authenticators using in-cache data. XOR MAC-SHA1 achieves 89% of the throughput of a system with no authentication.



(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 ratio of the file size by file size

Figure 4: Characterization of write I/Os from trace-driven experiments.

To better understand the run-time performance differences between XOR MAC and HMAC, we characterize the number and size of writes and how they are written to various files in the system. By looking at each write request as a function of its destination file size, we can see why incremental computation of MACs is beneficial to a file system. Our observations confirm three things: (1) Most write requests are small, (2) write requests are evenly distributed among all file sizes, and (3) the size of write requests are usually a tiny fraction of the file size. Figure 4(a) presents statistics on the number and size of write I/Os, whereas Figure 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 shows 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 writes 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.

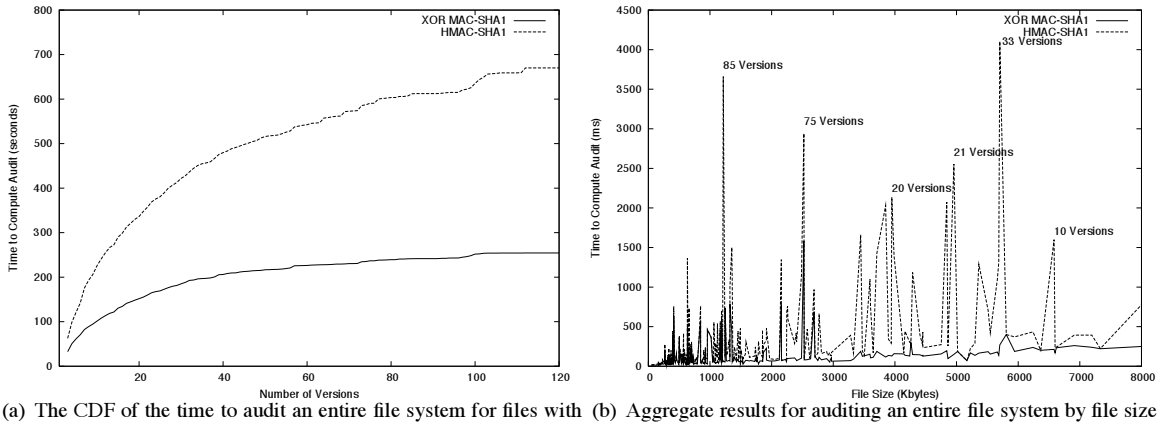
The relationship between I/O size and file size reveals the necessity of incremental MAC computation. Figure 4(c) presents the average write I/O size as a ratio of the file size over file sizes. This plot shows that there are few files that receive large writes or entire overwrites in a single I/O. In 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 XOR MAC. 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 XOR MAC, however, is immune to file size and is a function of write size alone.

6.2.2 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

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

Table 2: The number of seconds required to audit an entire file system using HMAC-SHA1 and XOR MAC-SHA1 for all files and only those files with two or more versions.



(a) The CDF of the time to audit an entire file system for files with two or more versions by number of versions

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

Figure 5: Aggregate auditing performance results for XOR MAC-SHA1 and HMAC-SHA1.

using XOR MAC-SHA1. Our audit calculated authenticators for every version of every file. Table 2 presents the aggregate results for performing an audit using XOR MAC-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 XOR MAC-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 XOR MAC 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 at auditing performance in the presence of versions, we filter out files with only one version. On files with two or more versions, XOR MAC-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 5(a). XOR MAC-SHA1 achieves a 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 XOR MAC performs.

Looking at audit performance by file size shows that the benefit is derived from long version chains. Figure 5(b) presents a break down of the aggregate audit results by file size. There exists no point at which XOR MAC-SHA1 performs worse than HMAC-SHA1, only points where they are the same or better. Performance is the same for files that have a single version and for files that do not share data among versions. 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. XOR MAC shows little performance variance with the number of versions.

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 are 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 modifications and number of authenticators generated were captured.

Figure 6 presents the size of authentication data generated over the simulation time for the three snapshot in-

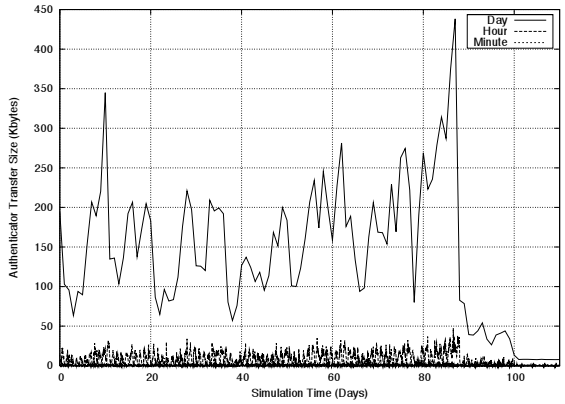


Figure 6: Size of authentication data from four months of traced workloads at three snapshot intervals.

tervals. Naturally, the longer the snapshot interval, the larger the number of authenticators generated. However, authentication data are 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 are 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 6 are identical to a figure graphing the number of files modified over the same snapshot intervals.

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.

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 architectures for managing authentication data; a storage-less third party and cooperative authentication. In a storage-less third party model 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 storing authenticators 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 file system root authenticator.

A further variant 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. 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.

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 [9, 18], such that accessing it in it's 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.* [8]. The system verifies authenticity while tolerating a small amount of modification, loss, or corruption of the original data.

We propose to make the AMAC construction incremental to adapt it to file systems; in addition, we plan to use XOR MAC as a building block in the AMAC construction [8], 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. The exact level of tolerance may be tuned.

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.

8 Conclusions

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. Experimental results show that incremental authentication can perform up to 94% faster than traditional serial authentication algorithms. We have implemented incremental authentication in ext3cow, an open-source versioning file system, available at: *www.ext3cow.com*.

9 Acknowledgments

This work was supported by the National Science Foundation (awards CCF-0238305 and IIS-0456027), by the Department of Energy, Office of Science (award DE-FG02-02ER25524), and by the IBM Corporation. We thank Giovanni Di Crescenzo for discussions on the AMAC construction.

References

[1] ADAMS, C., CAIN, P., PINKAS, D., AND ZUCCHERATO, R. IETF RFC 3161 Time-Stamp Protocol (TSP). IETF Network Working Group, 2001.

[2] AMERICAN BANKERS ASSOCIATION. American national standard for financial institution message authentication (wholesale). ANSI X9.9, 1986.

[3] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances in Cryptology - Crypto'96 Proceedings* (1996), vol. 1109, Springer-Verlag, pp. 1–19. Lecture Notes in Computer Science.

[4] BELLARE, M., GOLDREICH, O., AND GOLDWASSER, S. Incremental cryptography and application to virus protection. In *Proceedings of the ACM Symposium on the Theory of Computing* (May-June 1995), pp. 45–56.

[5] BELLARE, M., GUÉRIN, R., AND ROGAWAY, P. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *Advances in Cryptology - Crypto'95 Proceedings* (1995), vol. 963, Springer-Verlag, pp. 15–28. Lecture Notes in Computer Science.

[6] BLACK, J., AND ROGAWAY, P. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - Eurocrypt'02 Proceedings* (2002), vol. 2332, Springer-Verlag, pp. 384 – 397. Lecture Notes in Computer Science.

[7] BURNS, R., PETERSON, Z., ATENIESE, G., AND BONO, S. Verifiable audit trails for a versioning file system. In *Proceedings of the ACM CCS Workshop on Storage Security and Survivability* (November 2005), pp. 44–50.

[8] CRESCENZO, G. D., GRAVEMAN, R., GE, R., AND ARCE, G. Approximate message authentication and biometric entity authentication. In *Proceedings of Financial Cryptography and Data Security* (February-March 2005).

[9] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (October 2001), pp. 202–215.

[10] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael: AES – the Advanced Encryption Standard*. Springer, 2002.

[11] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX File and Storage Technologies Conference (FAST)* (March 2003), pp. 203–216.

[12] FARMER, D., AND VENEMA, W. *Forensic Disocvery*. Addison-Wesley, 2004.

[13] FU, K., KASSHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems 20*, 1 (2002), 1–24.

[14] GIFFORD, D. K., NEEDHAM, R. M., AND SCHROEDER, M. D. The Cedar file system. *Communications of the ACM 31*, 3 (March 1988), 288–298.

[15] HAUBERT, E., TUCEK, J., BRUMBAUGH, L., AND YURCIK, W. Tamper-resistant storage techniques for multimedia systems. In *IS&T/SPIE Symposium Electronic Imaging Storage and Retrieval Methods and Applications for Multimedia (EI121)* (January 2005), pp. 30–40.

[16] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. Information technology - security techniques - data integrity mechanism using a cryptographic check function employing a block cipher algorithm. ISO/IEC 9797, April 1994.

[17] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 29–42.

[18] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the ACM Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)* (November 2000), pp. 190–201.

[19] KUENNING, G. H., POPEK, G. J., AND REIHE, P. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer USENIX Technical Conference* (June 1994).

[20] LAMPORT, L. Password authentication with insecure comunication. *Communications of the ACM 24*, 11 (1981), 770–772.

[21] MANIATIS, P., AND BAKER, M. Enabling the archival storage of signed documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (January 2002), pp. 31–46.

[22] MCCOY, K. *VMS File System Internals*. Digital Press, 1990.

[23] MICALI, S. Efficient certificate revocation. Tech. Rep. MIT/LCS/TM-542b, Massachusetts Institute of Technology, 1996.

[24] MONROE, J. Emerging solutions for content storage. Presentation at PlanetStorage, 2004.

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

[26] MUNISWAMY-REDDY, K.-K., HOLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2006).

[27] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2004), pp. 115–128.

[28] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Digital signature standard (DSS). Federal Information Processing Standards (FIPS) Publication 186, May 1994.

[29] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Secure hash standard. Federal Information Processing Standards (FIPS) Publication 180-1, April 1995.

[30] PATIL, S., KASHYAP, A., SIVATHANU, G., AND ZADOK, E. I^3FS: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the Large Installation System Administration Conference (LISA)* (November 2004), pp. 67–78.

[31] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transcations on Storage 1*, 2 (2005), 190–212.

[32] PETERSON, Z. N. J., BURNS, R., HERRING, J., STUBBLEFIELD, A., AND RUBIN, A. Secure deletion for a versioning file system. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (December 2005), pp. 143–154.

[33] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (January 2002), pp. 89–101.

[34] RIVEST, R. L. All-or-nothing encryption and the package transform. In *Proceedings of the Fast Software Encryption Conference* (1997), vol. 1267, pp. 210–218. Lecture Notes in Computer Science.

[35] ROSELLI, D., AND ANDERSON, T. E. Characteristics of file system workloads. Research report, University of California, Berkeley, June 1996.

[36] ROSELLI, D., LORCH, J., AND ANDERSON, T. A comparison of file system workloads. In *Proceedings of the USENIX Technical Conference* (2000), pp. 41–54.

[37] SCHNEIER, B., AND KELSEY, J. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems Security 2*, 2 (1999), 159–176.

[38] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 43–58.

[39] UNITED STATES CONGRESS. The Health Insurance Portability and Accountability Act (HIPAA), 1996.

[40] UNITED STATES CONGRESS. The Sarbanes-Oxley Act (SOX). 17 C.F.R. Parts 228, 229 and 249, 2002.

[41] WALDMAN, M., RUBIN, A. D., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant, Web publishing system. In *Proceedings of the USENIX Security Symposium* (August 2000), pp. 59–72.

[42] WEATHERSPOON, H., WELLS, C., AND KUBIATOWICZ, J. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proceedings of the Workshop on Future Directions in Distributed Computing* (June 2002), pp. 142–147.