Verifiable Audit Trails for a Versioning File System

Randal Burns, Zachary Peterson, Giuseppe Ateniese, Stephen Bono
Department of Computer Science
The 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. 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. Our techniques address the I/O and computational efficiency of generating and verifying audit trails, the aggregation of audit information in directory hierarchies, and constructing verifiable audit trails in the presence of lost data.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Access methods, Directory Structure*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; K.5.2 [**Legal Aspects of Computing**]: Governmental Issues—*Regulation*; K.6.4 [**Management of Computing and Information Systems**]: System Management—*Management Audit*

General Terms

Algorithms, Management, Design, Security, Legal Aspects

Keywords

Secure audit, Electronic records, Versioning file systems

1. INTRODUCTION

The advent of Sarbanes-Oxley (SOX) [6] 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.

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 [26] 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 [21] or through a temporal query language [25]. However, for verification past versions of data must be immutable; writes modify the current version of a file, leaving the version history intact.

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 outline 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.

StorageSS'05, November 11, 2005, Alexandria, Virginia, USA.

Published data may even be stored publicly.

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 avoid all 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 [10]. We employ parallel message authentication codes [2, 3, 4] that allow MACs to be computed incrementally – based only on data that have changed from the previous version. MAC generation uses only data written in the cache, avoiding 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 [7]. This allows for a weaker statement of authenticity, but supports failure-prone storage environments.

2. 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 filesystem 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 timestamp 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 easily with a trusted third party, similar to a certificate authority. It may also be accomplished via publishing to censorship-resistant stores [27].

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 [9]. 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 [5].

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 not sufficient, because the storage contents may be read, modified, and written to a new WORM device.

Tamper-proof storage devices are a promising technology for the creation of immutable version histories [18]. 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 [11].

3. 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

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

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 [19]. As a corollary, K must be selected at random by the auditor. (The auditor adds randomness to the generation of K to meet the definition of a universal one-way hash function.) By including the file 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 [15], micropayments [23], certificate revocation [17], *etc.* 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. Although K is selected by the audi-

tor, it may be secret until audit. 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 [13], 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 timestamp [16]. 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 file 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 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.

3.1 Incrementally Calculable MACs

I/O efficiency is the principal concern in the calculation 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) [2, 3, 4] 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 [4].

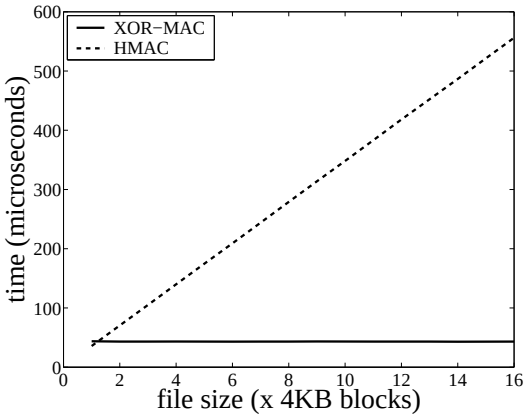

Figure 1: In-memory MAC computation

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 is divided into blocks $b_i(0), \ldots, b_i(n)$ equal to the file system block size. To be near consistent with the original publication [4], for block b_i, we label the one-way function on each block $Y(b_i)$. The output of the PMAC is the exclusive-or of the one-way functions of the input blocks and the previous version authenticator.

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

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.* $b_i(j) = b_{i-1}(j), j \neq k; b_i(k) \neq b_{i-1}(k)$, we observe that

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

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.

To demonstrate the benefit of incremental computation of MACs, we implemented a parallel MAC algorithm (XOR MAC [2]) and compare its performance to that of a hash MAC (HMAC) based on SHA-1. Figure 1 shows that running time of the algorithms when a single block of data changes for files of different sizes. Each data point represents the mean of 5000 trials conducted on a warm cache (all data in memory). This simple experiment confirms the known scaling properties of serial and parallel MACs. XOR MAC has more overhead on small files – 43 ms versus 35 ms

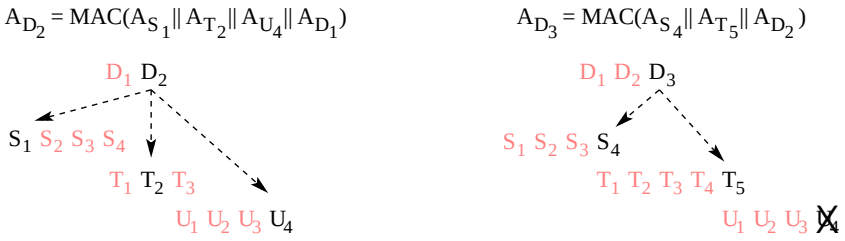

Figure 2: Updating directory version authenticators when file U is deleted.

for a single 4K block. However, parallel MAC performance scales with the amount of data changed: one block in this case. The performance of HMAC scales with the file size. This demonstrates the best case speedup for parallel MACs, when data change minimally between versions. However, studies of versioning file systems show that data change at a fine granularity [21, 26].

More importantly, the computation of the updated version authenticator may be performed on data available in the cache, requiring no additional disk I/O. In most cases, system caches are managed on a page basis, which leaves portions of any individual file version in memory and portions on the disk. This happens when files are not read in their entirety or previously read data are evicted from the cache (more likely with large or long-lived files). When computing a serial MAC for a file, all file data would need to be accessed, including that on disk. As disk accesses are a factor of 10^5 slower than memory accesses, computing a serial MAC is substantially worse than algorithmic performance would indicate. Only in the case of a blind write – write without reading – to a block not in cache does the incremental computation of a PMAC result in disk I/O.

3.2 Hierarchies and File Systems

Audit trails must include information about the entire file system; individual versions are not sufficient. 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.

Based on trees of MACs, we provide a construct that binds individual files to directories and to entire file systems. A directory version authenticator is a MAC of the version authenticators of its files and sub-directories along with the directory's previous version authenticator. Directory version authenticators continue recursively to the file system root, which is bound to the entire file system image. The SFS-RO system [10] 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 we must account for updates.

For efficiency reasons, we bind directory version authenticators to files (and sub-directories), not to specific versions of files. Figure 2 shows directory D_2 bound 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}. Rather, the system updates it only when the directory's contents change; *i.e.* files are created or destroyed.

The directory version authenticator binds the file and all subsequent versions of that file to the directory. The file (all versions) are part of the directory until deletion. In this example, when deleting file U the authenticator is updated to the current versions. We update directory version authenticators on file creation as well. Were we to bind directory version authenticators directly to the content of individual file versions, 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.

Updating directory version authenticators creates a time-space trade-off similar to that of publication frequency (Section 3). It is sufficient to update directory version authenticators when files are created or deleted only. However, updating them more frequently may be desirable to speed the audit process. During an audit, to verify that a file was in a directory at a particular point in time, the file must be included in a directory version authenticator prior to that point, to prove it existed, and after that point, to establish that it had not been deleted. Recall that the published history may not be complete and that the exact deletion time of a file may be unknown. However, the complete history may be reconstructed in an audit.

4. RELATED WORK

Most closely related to this work is the SFS-RO system [10], 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, distributed block. This detects corruption without relying on error correction and provides authenticity [28]. Patil *et al* [20] 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* [12] 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 [24]. 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: all changes to date. They 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.

5. 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 [8, 14], 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 to use approximately-secure and approximately-correct MAC (AMAC) introduced by Di Crescenzo et al. [7]. 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.

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 [7], 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.

6. FILE SYSTEM IMPLEMENTATION

We are implementing audit trails in ext3cow [21], 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-oriented interface. 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.

The ext3cow inode will be expanded to contain a authenticator for each file and directory version. 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. We have already retrofitted the metadata structures of ext3cow to support versioning and secure deletion (based on authenticated encryption [22]). Version authenticators are a straightforward extension to file system metadata, because they require only a few bytes per inode.

Key management in ext3cow uses lockboxes [13] to store a per-file authentication key, which the system generates automatically and stores within the inode. The file owner's private key unlocks the lockbox and provides access to the authentication key. Lockboxes are also part of the secure deletion feature of ext3cow [22].

7. 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 when data are created. This prevents the owner of the file system (or a malicious party) from modifying past data without detection. Algorithms for conducting audits rely on constructs with provable security. This includes techniques for the generation of audit metadata in parallel and methods to deal with data loss or temporary outages. We are currently implementing these technologies in a continuously versioning file system.

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. We also thank Peter Kimball for providing an implementation of the XOR MAC function used in Section 3.1.

8. REFERENCES

[1] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato. IETF RFC 3161 Time-Stamp Protocol (TSP). IETF Network Working Group, 2001.

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

[3] 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, Lecture Notes in Computer Science*, volume 963, pages 15–28. Springer-Verlag, 1995.

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

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

[6] United States Congress. Sarbanes-Oxley Act of 2002, 2002.

[7] 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*, 2005.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[9] D. Farmer and W. Venema. *Forensic Disocvery*. Addison-Wesley, 2004.

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

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

[12] E. Haubert, J. Tucek, L. Brumbaugh, and W. Yurcik. Tamper-resistant storage techniques for multimedia systems. In *International Symposium Electronic Imaging Storage and Retrieval Methods and Applications for Multimedia*, 2005.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the Conference on File and Storage Technologies*, 2003.

[14] 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 Conference on Architecture Support for Programming Languages and Operating Systems*, 2000.

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

[16] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proceedings of the Conference on File and Storage Technologies*, 2002.

[17] S. Micali. Efficient certificate revocation. In *Proceedings of RSA and US Patent 5,666,416*, 1997.

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

[19] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the Symposium on Theory of Computing*, 1989.

[20] 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*, 2004.

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

[22] 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*, 2004.

[23] R. L. Rivest and A. Shamir. PayWord and MicroMint – two simple micropayment schemes. *Proceedings of the International Workshop on Security Protocols, Lecture Notes in Computer Science, Springer*, (1189):69–87, 1997.

[24] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the USENIX Security Symposium*, 1998.

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

[26] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the Conference on File and Storage Technologies*, March 2003.

[27] 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*, 2000.

[28] 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*, 2002.

Appendix

The AMAC Construct (see [7]): 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 $\mathbf{Tag}(m, k, d)$ that computes the AMAC and returns the value tag, and a verification algorithm $\mathbf{Verify}(m, k, tag, d)$ that returns **true** if and only if $tag = \mathbf{Tag}(m, k, d)$.

An AMAC has $(d, p, \delta)-$*approximate-correctness* if $tag=\mathbf{Tag}(m, k, d)$, then with probability at least p $\mathbf{Verify}(m', k, tag, d)$ will return **true** if $d(m, m') \leq \delta$. An AMAC has $(d, \gamma, t, q, \epsilon)-$*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 $\mathbf{Verify}(m', k, tag, d)$ returns **true**, is at most ϵ.

Tag and Verify (cf. [7]): 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 [19].

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

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 3.1.
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.