Secure Deletion for a Versioning File System

Zachary N. J. Peterson Randal Burns Joe Herring
Adam Stubblefield Aviel D. Rubin
The Johns Hopkins University, Baltimore, MD

Abstract

We present algorithms and an architecture for the secure deletion of individual versions of a file. The principal application of this technology is federally compliant storage; it is designed to eliminate data after a mandatory retention period. However, it applies to any storage system that shares data between files, most notably versioning file systems. We compare two methods for secure deletion that use a combination of authenticated encryption and secure overwriting. We also discuss implementation issues, such as the demands that secure deletion places on version creation and the composition of file system metadata. Results show that new secure deletion techniques perform orders of magnitude better than previous methods.

1 Introduction

Versioning storage systems are increasingly important in research and commercial applications. Versioning has been recently identified by Congress as mandatory for the maintenance of electronic records of publicly traded companies (Sarbanes-Oxley, Gramm-Leach-Bliley), patient medical records (HIPAA), and federal systems (FISMA).

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

Secure deletion is valuable to security conscious users and organizations. It protects the privacy of user data and prevents the discovery of information on retired or sold computers. Traditional data deletion, or "emptying the trash", simply frees blocks for allocation at a later time; the data persists, fully readable and intact. Even when data are overwritten, information may be reconstructed using expensive forensic techniques, such as magnetic force microscopy [42].

We are particularly interested in using secure deletion to limit liability in the regulatory environment. By securely deleting data after they have fallen out of regulatory scope, *e.g.* seven years for corporate records in Sarbanes-Oxley, data cannot be recovered even if disk drives are produced and encryption keys revealed. Data are gone forever and corporations are not subject to exposure via subpoena or malicious attack.

Currently, there are no efficient methods for fine-grained secure deletion in storage systems that share data among files, such as versioning file systems [12, 20, 27, 25, 32, 35] and content-indexing systems [2, 26, 28].

The preferred and accepted methods for secure deletion in non-data sharing systems include: repeatedly overwriting data, such that the original data may not be recovered [17]; and, encrypting a file with a key and securely disposing of the key to make the data unrecoverable [8].

Block sharing hinders key management in encrypting systems that use key disposal. If a system were to use an encryption key per version, the key could not be discarded, as it is needed to decrypt shared blocks in future versions that share the encrypted data. To realize fine-grained secure deletion by key disposal, a system must keep a key for every shared block, resulting in an onerous number of keys that quickly becomes unmanageable. Fewer keys allow for more flexible security policies [22].

Secure overwriting also has performance concerns in versioning systems. In order to limit storage overhead, versioning systems often share blocks of data between file versions. Securely overwriting a shared block in a past version could erase it from subsequent versions. To address this, a system would need to detect data sharing dependencies among all versions before committing to a deletion. Also, in order for secure overwriting to be efficient, the data to be removed should be contiguous on

disk. Non-contiguous data blocks require many seeks by the disk head – the most costly disk drive operation. By their very nature, versioning systems are unable to keep the blocks of a file contiguous in all versions.

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

To our knowledge, we are the first file system to adopt authenticated encryption (AE) [4], which provides both privacy and authenticity. Authenticity is essential to ensure that the data have not changed between being written to disk and read back. Particularly in environments where storage is virtualized or distributed and, thus, difficult to physically secure. Authenticated encryption requires message expansion – ciphertext are larger than the plaintext – which is an obstacle to its adoption. Encrypting file systems have traditionally used block ciphers, which preserve message size, to meet the alignment and capacity constraints of disk drives [5, 40, 22]. In practice, additional storage must be found for the expanded bits of the message. Our architecture creates a parallel structure to the inode block map for the storage of expanded bits of the ciphertext and leverages this structure to achieve secure deletion. Message expansion is fundamental to our deletion model.

We have implemented secure deletion and authenticated encryption in the ext3cow versioning file system, designed for version management in the regulatory environment [27]. Experimental results show that our methods for secure deletion improve deletion performance by several orders of magnitude. Also, they show that metadata maintenance and cryptography degrade file system performance minimally.

2 Related Work

Secure Deletion

Garfinkel and Shelat [16] survey methods to destroy digital data. They identify secure deletion as a serious and pressing problem in a society that has a high turnover in technology. They cite an increase in lawsuits and news reports on unauthorized disclosures, which they attribute to a poor understanding of data longevity and a lack of secure deletion tools. They identify two methods of secure deletion that leave disk drives in a usable condition: secure overwriting and encryption.

In secure overwriting, new data are written over old data so that the old data are irrecoverable. Gutmann [17] gives a technique that takes 35 synchronous passes over the data in order to degauss the magnetic media, making the data safe from magnetic force microscopy. (Fewer passes may be adequate [16]). This technique has been implemented in user-space tools and in a Linux file system [3]. Secure overwriting has also been applied in the semantically-smart disk system [34].

For file systems that encrypt data on disk, data may be securely deleted by "forgetting" the corresponding encryption key [8]; without a key, data may never be decrypted and read again. This method works in systems that maintain an encryption key per file and do not share data between multiple files. The actual disposal of the encryption key may involve secure overwriting.

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

The importance of deleting data has been addressed in other system components. A concept related to stub deletion has been used in memory systems [13], which erase a large segment of memory by destroying a small non-volatile segment. Securely deallocating memory limits the exposure of sensitive data [11]. Similar problems have been addressed by Gutmann [18, 19] and Viega [37].

Secure Systems

CFS [5] was an early effort that added encryption to a file system. In this user-space tool, local and remote (via NFS) encrypted directories are accessed via a separate mount point. All file data and metadata in that directory are encrypted using a pre-defined user key and encryption algorithm. CFS does not provide authenticated encryption.

NCryptfs [40] is a cryptographic file system implemented as a stackable layer in FiST [41]. The system is designed to be customizable and flexible for its users by providing many options for encryption algorithms and key requirements. It does not provide authenticated encryption.

Cryptoloop uses the Linux cryptographic API [24] and the loopback interface to provide encryption for blocks as they are passed through to the disk. While easy to ad-

minister for a single-user machine, cryptographic loopback devices do not scale well to multi-user systems.

Our implementation of encryption follows the design of the CryptoGraphic Disk Driver (CGD) [15]. CGD replaces the native disk device driver with one that encrypts blocks as they are transfered to disk.

The encryption and storage of keys in the random-key encryption scheme resembles lock-boxes in the Plutus file system [22] in which individual file keys are stored in lock-boxes and sealed with a user's key.

Cryptography

Secure deletion builds upon cryptographic constructs that we adapt to meet the demands of a versioning file system. The principal methods that we employ are the all-or-nothing transform [29], secret-sharing [33], and authenticated encryption [4]. Descriptions of their operation and application appear in the appropriate technical sections.

3 Secure Deletion with Versions

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

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

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

that takes a data block (B_i), a key (k) and a nonce (N) and creates an output that can be partitioned into an encrypted data block (C_i), where $|B_i| = |C_i|$, and a short *stub* (s_i), whose length is a parameter of the scheme's security. When the key (k) remains private, the transform acts as an authenticated encryption algorithm. To securely delete an entire block, only the stub needs to be securely overwritten. This holds *even if the adversary is later given the key (k),* which models the situation in which a key is exposed, *e.g.* by subpoena. The stub reveals nothing about the key or the data, and, thus, stubs may be stored on the same disk. It may be possible to recover securely deleted data after the key has been exposed by a brute-force search for the stub. However, this is no easier than a brute-force search for a secret key and is considered intractable.

A distinct advantage of our file system architecture is the use of authenticated encryption [4]. Authenticated encryption is a transform by which data are kept both private *and* authentic. Many popular encryption algorithms, such as AES, by themselves, provide only privacy; they cannot guarantee that the decrypted plaintext is the same as the original plaintext. When decrypting, an authenticated encryption scheme will take a ciphertext and return either the plaintext or an indication the ciphertext is invalid or unauthentic. A common technique for authenticated encryption is to combine a message authentication code (MAC) with a standard block cipher [4]. However, single pass methods exist [30].

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

Our architecture for secure deletion with stubs does not complicate key management. It employs the same key-management framework used by disk-encrypting file systems based on block ciphers, such as Plutus [22] and NCryptfs [40]. It augments these to support authenticated encryption and secure deletion.

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

3.1 AON Secure Deletion

The all-or-nothing (AON) transform is a cryptographic function that, given a partial output, reveals nothing about its input. No single message of a ciphertext can be decrypted in isolation without decrypting the entire ciphertext. The transform requires no additional keys. The original intention, as proposed by Rivest [29], was to prevent brute-force key search attacks by requiring the attacker to decrypt an entire message for each key guess, multiplying the work by a factor of the number of blocks in the message. Boyko presented a formal definition for the AON transform [9] and showed that the OAEP [4] scheme used in many Internet protocol standards meets his definition. AON has been proposed to make efficient smart-card transactions [6, 7, 21], message authentication [14], and threshold-type cryptosystems using symmetric primitives [1].

The AON transform is the most natural construct for

Input: Data Block $d_1, \ldots, d_m$, Block ID id, Counter x, Encryption key K, MAC key M

1: $ctr_1 \leftarrow id||x||1||0^{128-|x|-|id|-1}$
2: $c_1, \ldots, c_m \leftarrow \text{AES-CTR}_K^{ctr_1}(d_1, \ldots, d_m)$
3: $t \leftarrow \text{HMAC-SHA-1}_M(c_1, \ldots, c_m)$
4: $ctr_2 \leftarrow id||x||0||0^{128-|x|-|id|-1}$
5: $x_1, \ldots, x_m \leftarrow \text{AES-CTR}_t^{ctr_2}(c_1, \ldots, c_m)$
6: $x_0 \leftarrow x_1 \oplus \ldots \oplus x_m \oplus t$

Output: Stub x_0, Ciphertext $x_1, \ldots, x_m$

(a) AON encryption

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

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

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

(b) AON decryption

Figure 1: Authenticated encryption and secure deletion for a single data block in a versioning file system using the all-or-nothing scheme.

the secure deletion of versions. We aim to minimize the amount of secure overwriting. We also aim to not complicate key management. AON fulfills both requirements while conforming to our deletion model. The all-or-nothing property of the transform allows the system to overwrite any small subset of a data block to delete the entire block; without all subsets, the block cannot be read. When combined with authenticated encryption, the AON transform creates a message expansion that is bound to the same all-or-nothing property. This expansion is the stub and can be securely overwritten to securely delete a block. Because the AON transform requires no additional keys, key management is no more complicated than a system that uses a block cipher.

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

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

Despite the virtues of providing authenticated encryption with low performance and storage overheads, this construction of AON encryption suffers from a guessed-plaintext attack. After an encryption key has been revealed, if an attacker can guess the exact contents of a block of data, she can verify that the data were once in the file system. This attack does not reveal encrypted data. Once the key is disclosed, the attacker has all of the inputs to the encryption algorithm and may reproduce the ciphertext. The ciphertext may be compared to

Input: Data Block $d_1, \ldots, d_m$, Block ID id, Counter x, Encryption key K, MAC key M

1: $k \xleftarrow{R} \mathcal{K}_{AE}$
2: $nonce \leftarrow id||x$
3: $c_1, \ldots, c_n \leftarrow \text{AE}_k^{nonce}(d_1, \ldots, d_m)$
4: $ctr \leftarrow id||x||0^{128-|x|-|id|}$
5: $c_0 \leftarrow \text{AES-CTR}_K^{ctr}(k)$
6: $t \leftarrow \text{HMAC-SHA-1}_M(ctr, c_0)$

Output: Stub $c_0, t, c_{m+1}, \ldots, c_n$, Ciphertext $c_1, \ldots, c_m$

(a) Random-key encryption

Input: Stub $c_0, t, c_{n+1}, \ldots, c_m$, Ciphertext $c_1, \ldots, c_n$, Block ID id, Counter x, Encryption key K, MAC key M

1: $ctr \leftarrow id||x||0^{128-|x|-|id|}$
2: $t' \leftarrow \text{HMAC-SHA-1}_M(ctr, c_0, r)$
3: if $t' \neq t$ return $\perp$
4: $k \leftarrow \text{AES-CTR}_K^{ctr}(c_0)$
5: $nonce \leftarrow id||x$
6: $d_1, \ldots, d_n = \text{AE}_k^{nonce}(c_1, \ldots, c_m)$

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

(b) Random-key decryption

Figure 2: Authenticated encryption and secure deletion for a single data block in a versioning file system using the random-key scheme.

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

Such an attack is reasonable within the threat model of regulatory storage; a key may be subpoenaed in order to show that the file system contained specific data at some time. For example, to show that an individual had read and subsequently made attempts to destroy an incriminating email. This threat can be eliminated by adding a random postfix to the data block, though this will increase the size of the stub.

3.2 Secure Deletion Based on Randomized Keys

As mentioned by Rivest [29], avoiding such a text-guessing attack requires that an AON transform employ randomization so that the encryption process is not repeatable given the same inputs. The subsequent security construct generates a random key on a per-block basis.

Random-key encryption is not an all-or-nothing transform. Instead, it is a refinement of the Boneh key disposal technique [8]. Each data block is encrypted using a randomly generated key. When this randomly generated key is encrypted with the file key, it acts as a stub. Like AON encryption, random-key encryption makes use of authenticated encryption, minimizes the amount of data needed to be securely overwritten, and does not require the management of additional keys.

We give an algorithm for random-key secure deletion in Figure 2. To encrypt (Figure 2(a)), the scheme generates a random key, k, (Step 1) that is used to authenticate and encrypt a data block. Similar to the unique counters in the AON scheme, a unique nonce is generated (Step 2). Data is then encrypted and authenticated (Step 3), resulting in an expanded message. The algorithm is built upon any authenticated encryption (AE) scheme; AES and SHA-1 satisfy standard security definitions. To avoid the complexities of key distribution, we employ a single encryption (K) and MAC (M) key per file (the same keys as used in AON encryption) and use these keys to encrypt and authenticate the random key (k) (Step 5). The encrypted randomly-generated key (c_0) serves as the stub. The expansion created by the AE scheme in Step 3 ($c_{m+1}, \ldots, c_n$), and the authentication of the encrypted random key (t) does not need to be securely overwritten to permanently destroy data.

An advantage of random-key encryption over AON encryption is its speed. For example, when the underlying AE is OCB [30], only one pass over the data is made and it is fully parallelizable. However, the algorithm suffers from a larger message expansion: 384 bits per disk block are required instead of 128 required for the AON scheme. We are exploring other more space-efficient algorithms. We have developed another algorithm that requires no more bits than the underlying AE scheme. Unfortunately, this is based on OAEP and a Luby-Rackoff construction [23] and is only useful for demonstrating that space efficient constructions do exist. It is far too slow to be used in practice, requiring six expensive passes over the data.

3.3 Other Secure Deletion Models

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

3.3.1 Deleting a Version Chain

AON encryption allows for the deletion of a block of data from an entire version chain. Due to the all-or-nothing properties of the transform, the secure overwriting of *any* 128 bits of a data block results in the secure deletion of that block, even if the stub persists. When a user wishes to delete an entire version chain, *i.e.* all blocks associated with all versions of a file, it may be more efficient to securely overwrite the blocks themselves rather than each version's stubs. This is because overwriting is slow and many blocks are shared between versions. For example, to delete a large log file to which data has only been appended, securely deleting all the blocks in the most recent version will delete all past versions. Ext3cow provides a separate interface for securely deleting data blocks from all versions. If a deleted block was shared, it is no longer accessible to other versions, despite their possession of the stub.

Randomized-key encryption does not hold this advantage; only selective components may be deleted, *i.e.* c_0. Thus, in order to delete a block from all versions, the system must securely overwrite all stub occurrences in a version chain, as opposed to securely overwriting only 128 bits of a data block in an AON scheme. To remedy this, a key share (Section 3.3.2) could be stored alongside the encrypted data block. When the key share is securely overwritten, the encrypted data are no longer accessible in any version. However, this strategy is not practical in most file systems, owing to block size and alignment constraints. Storage for the key share must be provided and there is no space in the file system block. The shares could be stored elsewhere, as we have with deletion stubs, but need to be maintained on a per-file, rather than per-version, basis.

3.3.2 Secure Deletion with Secret-Sharing

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

Our random-key encryption scheme allows for the separation of the randomly-generated encryption key into n key shares. This is a form of an (m, n) secret-sharing scheme [33]. In secret-sharing, Shamir shows how to divide data into n shares, such that any m shares can reconstruct the data, but where $m - 1$ shares reveals nothing about the data. We are able to compose a single randomly generate encryption key (k) from multiple key shares. An individual key share may then be given to a user with an interest in the data, distributing the means to delete data. If a single key share is independently deleted, the corresponding data are securely deleted and the remaining key shares are useless. Without all key shares, the randomly generated encryption key may not be reconstructed and decryption will fail.

Any number of randomly generated keys may be created in Step 1 (Figure 2(a)) and composed to create a single encryption key (k). To create two key shares (a $(2, 2)$ scheme), Step 1 may be replaced with:

$$\ell, r \xleftarrow{R} \mathcal{K}_{AE}$$
$$k \leftarrow \ell \oplus r$$

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

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

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

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

3.4 Security Properties

Confidence is gained in modern cryptographic constructions through the use of reductionist arguments: it is shown that if an adversary can break a particular construction, he can also break the underlying primitives that are employed. For example, AES in CTR mode can be shown to be secure so long as the AES algorithm is itself secure.

As was previously pointed out, the authenticated AON scheme is not secure as it falls victim to a plaintext guessing attack. Even if this particular problem is fixed (by appending a random block to the plaintext, thereby increasing the size of the stub), the construction is not necessarily secure. Due to some technical problems with the model, a proof that this type of "package transform" construction reduces to the security of the underlying block cipher has eluded cryptographers for several years.

The random keys construction is provably secure (under reasonable definitions for this application) so long

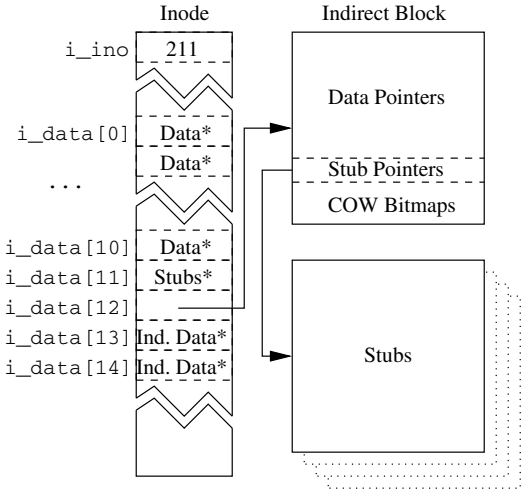


Figure 3: Metadata architecture to support stubs.

as the underlying authenticated encryption scheme is secure, AES is secure, HMAC-SHA1 is secure, and SHA-1 acts as a random oracle. We omit the formal definitions and proofs from this work.

4 Architecture

We have implemented secure deletion in ext3cow [27], an open-source, block-versioning file system designed to meet the requirements of electronic record management legislation. Ext3cow supports file system snapshot, per-file versioning, and a time-shifting interface that provides real-time access to past versions. Versions of a file are implemented by chaining inodes together where each inode represents a version of a file.

4.1 Metadata for Secure Deletion

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

A similar "block stealing" design was chosen for managing stubs. A number of block addresses in the inode and the indirect blocks have been reserved to point to blocks of stubs. Figure 3 illustrates the metadata architecture. The number of direct blocks in an inode has been reduced by one, from twelve to eleven, for storage of stubs (`i_data[11]`) that correspond to the direct blocks. Ext3cow reserves words in indirect blocks to be used as pointers to blocks of stubs.

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

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

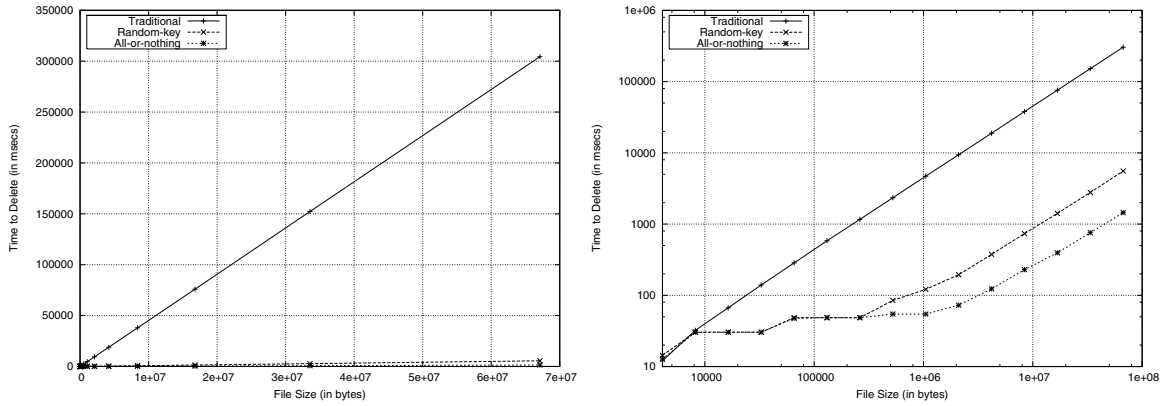
Because the extra metadata borrows space from indirect blocks, the design reduces the maximum file size. The loss is about 16%. With a 4K block size, ext3cow represents files up to 9.03×10^8 blocks in comparison to 1.07×10^9 blocks in ext3. The upcoming adoption of quadruply indirect blocks by ext3 [36] will remove practical file size limitations.

4.2 The Secure Block Device Driver

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

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

Cryptography in the device driver was built upon the pre-existing cryptographic API available in the Linux



(a) Time to Delete (linear)

(b) Time to Delete (log-log)

Figure 4: The time to securely delete files for the secure overwriting (traditional), all-or-nothing, and random-key techniques.

kernel [24], namely the AES and SHA-1 algorithms. Building upon existing constructs simplified development, and aids correctness. Further, it allows for the security algorithms to evolve, giving opportunity for the secure deletion transforms to be updated as more secure algorithms become available. For instance, the security of SHA-1 has been recently called into question [38].

We plan to release the secure device driver to the Linux community under an open source license via the ext3cow website, *www.ext3cow.com*.

4.3 Security Policies

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

With secure deletion, I/O drives the creation of versions. Our architecture mandates a new version whenever a block and a stub are written to disk. Continuous versioning, *e.g.* CVFS [35], meets this requirement, because it creates a new version on every `write()` system call. However, for many users, continuous versioning may incur undesirable storage overheads, approximately 27% [27, 35]. Most systems create versions less frequently. As a matter of policy, *e.g.* daily, on every file open, *etc.*; or, explicitly through a snapshot interface.

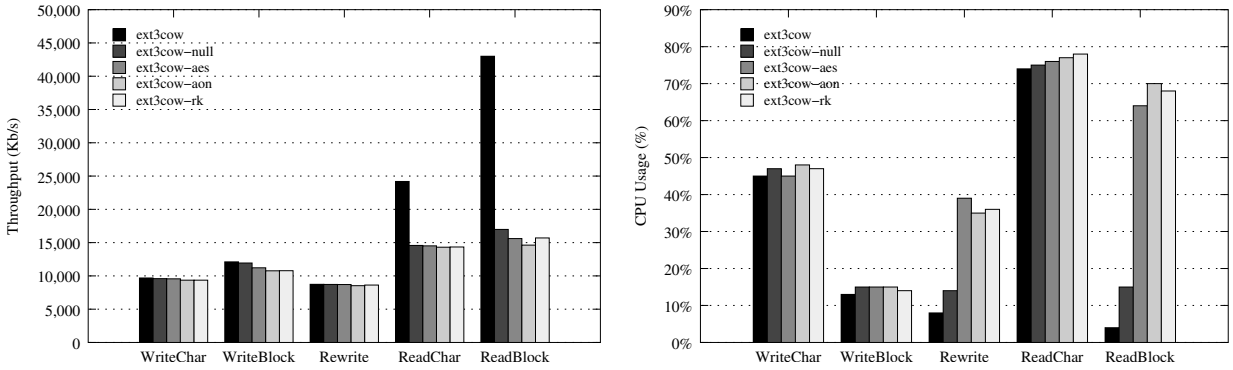
The demands of secure deletion may be met without continuous versioning. Ext3cow reduces the creation of versions based on the observation that multiple writes to the same stub may be aggregated in memory prior to reaching disk. We are developing write-back caching policies that delay writes to stub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

5 Experimental Results

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

5.1 Time to Delete

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



(a) Throughput

(b) CPU Utilization

Figure 5: Bonnie++ throughput and CPU utilization results.

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

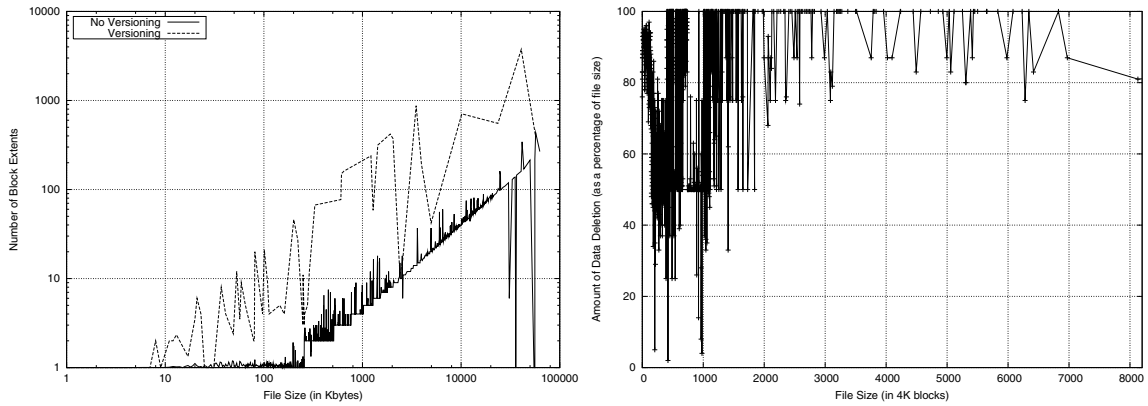
Figure 4(a) demonstrates the dramatic savings in time that can be achieved by using stub deletion. Files between 2^{16} and 2^{20} were truncated for clarity. AON deletion bests traditional deletion by a factor of 200 for 67MB files (2^{15} blocks), with random-key deletion performing slightly worse than AON. Differences are better seen in Figure 4(b), a log-log plot of the same result.

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

5.2 Bonnie++

Bonnie++ is a well-known performance benchmark that quantifies five aspects of file system performance based on observed I/O bottlenecks in a UNIX-based file system. Bonnie++ performs I/O on large files (for our experiment, two 1-GB files) to ensure I/O requests are not served out of the disk's cache. For each test, Bonnie++ reports throughput, measured in kilobytes per second, and CPU utilization, as a percentage of CPU usage. Five operations are tested: (1) each file is written sequentially by character, (2) each file is written sequentially by block, (3) the files are sequentially read and rewritten, (4) the files are read sequentially by character, and (5) the files are read sequentially by block. We compare the results of five file system modes: ext3cow, ext3cow-null, ext3cow-aes, ext3cow-aon and ext3cow-rk. Respectively, they are: a plain installation of ext3cow with no secure device driver. Ext3cow with a secure device driver that does no encryption. Ext3cow with a secure device driver that does a simple AES encryption. Ext3cow with a secure device driver that runs the all-or-nothing algorithm, and ext3cow with a secure device driver that runs the random-key algorithm. Ext3cow performs comparably with ext3 [27]. Results are the product of an average of 10 runs of Bonnie++ on the same partition.

Figure 5(a) presents throughput results for each Bonnie++ test. When writing, throughput suffers very little in the presence of cryptography. The largest difference occurs when writing data a block at a time; AON encryption reduces throughput by 1.3 MB/s, from 12.1 MB/s to 10.8 MB/s. This result is consistent with the literature [39]. A more significant penalty is incurred when reading. However, we believe this to be an artifact of the driver and not the cryptography, as the null driver (the secure device driver employing no cryptography) experiences the same performance deficit. The problem stems from the secure device driver's inability to aggregate local block requests into a single large request. We are currently implementing a request clustering algorithm that will eliminate the disparity. In the meantime, the differences in the results for the null device driver and device drivers that employ cryptography are minor: a maximum difference of 200 K/s for character reading and 1.2 MB/s for block reading. Further, the reading of stubs has no effect on the ultimate throughput. We attribute this to ext3cow's ability to co-locate stubs with the data they represent. Because it is based on ext3 [10], ext3cow employs block grouping to keep metadata and



(a) Number of Block Extents

(b) Amount of Data to Delete

Figure 6: Results of trace-driven file system aging experiments.

data near each other on disk. Thus, track caching on disk and read-ahead in ext3cow put stubs into the disk and system cache, making them readily available when accessing the corresponding data.

To gauge the impact of file system cryptography on the CPU, we measured the CPU utilization for each Bonnie++ test. Results are presented in Figure 5(b). When writing, cryptography, as a percentage of the total CPU, has nearly no effect. This makes sense, as more of the CPU is utilized by the operating system for writing than for reading. Writes may perform multiple memory copies, allocate memory pages, and update metadata. Similarly, reading data character by character is also CPU intensive, due to buffer copying and other memory management operations, so cryptography has a negligible effect. Cryptography does have a noticeable effect when reading data a block at a time, evident in the rewrite and block read experiments. Because blocks match the page size in ext3cow, little time must be spent by the CPU to manage memory. Thus, a larger portion of CPU cycles are spent on decryption. However, during decryption, the system remains I/O bound, as the CPU never reaches capacity. These results are consistent with recent findings [39] that the overheads of cryptography are acceptable in modern file systems.

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

5.3 Trace-Driven Experiments

We present results that quantify the difficulty of achieving good performance when securely deleting data that have fallen out of regulatory scope. We replayed four months of file system call traces [31] on an 80G ext3cow partition, taking snapshots every second. This results in 4.2 gigabytes of data in 81674 files.

We first examine the amount of external fragmentation that results from versioning. External fragmentation is the phenomenon of file blocks in nonadjacent disk addresses. This causes multiple disk drive seeks to read or delete a file. Ext3cow uses a copy-on-write scheme to version files [27]. This precludes the file system from keeping all blocks of a version strictly contiguous. Because seeks are an expensive operations, fragmentation is detrimental to the performance of traditional secure overwriting. Figure 6(a) shows the effect versioning has on block fragmentation. Versioning increases dramatically the average number of block extents – regions of contiguous blocks. This is in comparison to the ext3 file system without versioning. Note the log-log scale. Some files have as many as 1000 block extents. This is the result of files receiving lots copy-on-write versioning.

In practice, secure deletion provides more benefit than microbenchmark results would indicate (Section 5.1). Given that seeking is the most expensive disk operation, traditional secure overwriting scales with the number of block extents that need to be overwritten. For AON or random-key secure deletion, the number of extents depends only upon the file size, not the fragmentation of data. Deletion performance does not degrade with versioning. For secure overwriting of the file data, performance scales with the number of block extents. Given the large degree of fragmentation generated through ver-

sioning, isolating deletion performance from file contiguity is essential.

Despite the high degree of copy-on-write and fragmentation, trace results show that there are considerable data to delete in each version, *i.e.* deletion is non-trivial. When a version of a file falls out of scope, much of its data are unique to that version and, thus, need to be securely deleted. This is illustrated in Figure 6(b). This graph shows the average amount of data that needs to be deleted as a percentage of the file size. There are very few files that have fewer than 25% unique blocks. Most versions need 100% of their blocks deleted. This is not unexpected as many files are written once and never modified. This is much more important for larger files which are more sensitive to deletion performance; stub deletion offers less benefit when deleting very small files. Even the largest files in the file system contain mostly unique data.

6 Applicability to Other Data System

There is potential for the reuse of the AON and random-key algorithms for secure deletion in any storage system that shares data among files. Content-indexing systems, such as Venti [28], LBFS [26], and pStore [2], have the same deletion problems and our technology translates directly. Content-indexing stores a corpus of data blocks (for all files) and represents a file as an assemblage of blocks in the corpus. Files that share blocks in the corpus have the same dependencies as do copy-on-write versions.

7 Conclusions

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

We present two algorithms within this model. The first algorithm employs the all-or-nothing transform so that securely overwriting a stub or any 128 bit block of a ciphertext securely deletes the corresponding disk block. The second algorithm generates a random key per block in order to make encryption non-repeatable. The first algorithm produces more compact stubs and supports a richer set of deletion primitives, whereas the second algorithm provides stronger privacy guarantees.

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

8 Acknowledgments

This work was supported in part by NSF award CCF-0238305, by the DOE Office of Science award P020685, the IBM Corporation, and by NSF award IIS-0456027.

References

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

[2] BATTEN, C., BARR, K., SARAF, A., AND TREPETIN, S. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[3] BAUER, S., AND PRIYANTHA, N. B. Secure data deletion for Linux file systems. In *Proceedings of the USENIX Security Symposium* (August 2001).

[4] BELLARE, M., AND NAMPREMPRE, C. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt'00 Proceedings* (2000), vol. 1976, Springer-Verlag. Lecture Notes in Computer Science.

[5] BLAZE, M. A cryptographic file system for UNIX. In *Proceedings of the ACM conference on Computer and Communications Security* (November 1993), pp. 9–16.

[6] BLAZE, M. High-bandwidth encryption with low-bandwidth smartcards. In *Fast Software Encryption* (1996), vol. 1039, pp. 33–40. Lecture Notes in Computer Science.

[7] BLAZE, M., FEIGENBAUM, J., AND NAOR, M. A formal treatment of remotely keyed encryption. In *Advances in Cryptology – Eurocrypt'98 Proceedings* (1998), vol. 1403, pp. 251–265. Lecture Notes in Computer Science.

[8] BONEH, D., AND LIPTON, R. A revocable backup system. In *Proceedings of the USENIX Security Symposium* (July 1996), pp. 91–96.

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

[10] CARD, R., TS'O, T. Y., AND TWEEDIE, S. Design and implementation of the second extended file system. In *Proceedings of the Amsterdam Linux Conference* (1994).

[11] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium* (August 2005), pp. 331–346.

[12] CORNELL, B., DINDA, P. A., AND BUSTAMANTE, F. E. Wayback: A user-level versioning file system for Linux. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2004), pp. 19–28.

[13] CRESCENZO, G. D., FERGUSON, N., IMPAGLIAZZO, R., AND JAKOBSSON, M. How to forget a secret. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science* (1999), vol. 1563, Springer-Verlag, pp. 500–509. Lecture Notes in Computer Science.

[14] DODIS, Y., AND AN, J. Concealment and its applications to authenticated encryption. In *Advances in Cryptology – Eurocrypt'03 Proceedings* (2003), vol. 2656. Lecture Notes in Computer Science.

[15] DOWDESWELL, R., AND IOANNIDIS, J. The CryptoGraphic disk driver. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2003), pp. 179–186.

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

[17] GUTMANN, P. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the USENIX Security Symposium* (July 1996), pp. 77–90.

[18] GUTMANN, P. Software generation of practically strong random numbers. In *Proceedings of the USENIX Security Symposium* (January 1998), pp. 243–257.

[19] GUTMANN, P. Data remanence in semiconductor devices. In *Proceedings of the USENIX Security Symposium* (August 2001), pp. 39–54.

[20] HITZ, D., LAU, J., AND MALCOM, M. File system design for an NFS file server appliance. In *Proceedings of the Winter USENIX Technical Conference* (January 1994), pp. 235–246.

[21] JAKOBSSON, M., STERN, J., AND YUNG, M. Scramble all. Encrypt small. In *Fast Software Encryption* (1999), vol. 1636. Lecture Notes in Computer Science.

[22] 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.

[23] LUBY, M., AND RACKOFF, C. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing 17*, 2 (April 1988), 373–386.

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

[25] 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.

[26] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (October 2001), pp. 174–187.

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

[28] 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.

[29] 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.

[30] ROGAWAY, P., BELLARE, M., BLACK, J., AND KROVET, T. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proceedings of the ACM Conference on Computer and Communications Security* (November 2001), pp. 196–205.

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

[32] SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (December 1999), pp. 110–123.

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

[34] SIVATHANU, M., BAIRAVASUNDATAM, L., ARPACI-DUSSAEU, A. C., AND ARPACI-DUSSEAU, R. H. Life or Death at Block-Level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004), pp. 379–394.

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

[36] TS'O, T. Y., AND TWEEDIE, S. Planned extensions to the Linux ext2/ext3 filesystem. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2002), pp. 235–243.

[37] VIEGA, J., AND MCGRAW, G. *Building Secure Software*. Addison-Wesley, 2002.

[38] WANG, X., YIN, Y. L., AND YU, H. Finding collisions in the full SHA-1. In *Advances in Cryptology - Crypto'05 Proceedings* (August 2005), Springer-Verlag. Lecture Notes in Computer Science. To appear.

[39] WRIGHT, C., DAVE, J., AND ZADOK, E. Cryptographic file systems performance: What you don't know can hurt you. In *Proceedings of the IEEE Security in Storage Workshop (SISW)* (October 2003), pp. 47–61.

[40] WRIGHT, C. P., MARTINO, M., AND ZADOK, E. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the USENIX Technical Conference* (June 2003), pp. 197–210.

[41] ZADOK, E., AND NIEH, J. FiST: A language for stackable file systems. In *Proceedings of the USENIX Technical Conference* (June 2000), pp. 55–70.

[42] ZHU, J.-G., LUO, Y., AND DING, J. Magnetic force microscopy study of edge overwrite characteristics in thin film media. *IEEE Transaction on Magnetics 30*, 6 (1994), 4242–4244.