Secure Deletion in a Versioning File System

Zachary N. J. Peterson

Qualifying Project
Computer Science Department
The Johns Hopkins University

Abstract

We present an architecture for the secure deletion of individual versions of a file. The principle application of this technology is federally compliant storage; it is designed to eliminate data after a mandatory retention period. However, it applies generally to any storage system that shares data between files, most notably versioning file systems and content-indexing archives. We compare two methods for encrypting and encoding data. We also discuss implementation issues, such as the demands that secure deletion places on version creation and the composition of file system metadata.

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 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. 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, their are no efficient methods for fine-grained secure deletion in versioning storage systems. The preferred and accepted methods for secure deletion in non-versioning systems include: overwriting data with other data, such that the original data may not be recovered [6]; and, encrypting a file with a key and securely disposing of the key to make the data unrecoverable [3]. These techniques are not applicable to versioning systems.

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

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

We have developed 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. For 4K, blocks, this is a 256 times speedup. Further, we collect and store stubs contiguously in a file system block so that overwriting a 4K block of stubs deletes the corresponding 1 MB of file data, even when file data are non-contiguous. Unlike encryption keys, stubs are not secret and may be stored on disk. We are implementing these deletion techniques in the ext3cow versioning file system, designed for version management in the regulatory environment [9].

2 Related Work

Garfinkel and Shelat [5] give a survey of methods to destroy digital data. They identify secure deletion as a serious and pressing problem in a society that has a high turn-over in technology. They cite an increase in law suits and news reports on unauthorized disclosures, which they attribute to a poor understanding of data longevity and a lack of secure deletion tools. They identify two methods of secure deletion that leave disk drives in a usable condition: secure overwriting and encryption.

In secure overwriting, new data are written over old data so that the old data are irrecoverable. Gutmann [6] gives a technique that takes 22 synchronous passes over the data in order to degauss the magnetic media, making the data safe from magnetic force microscopy. (Fewer passes may be adequate [5]). This has been implemented in user-space tools and in a Linux file system [1]. Secure overwriting has also been applied in semantically-smart disk systems [13]. However, a large number of synchronous passes may be prohibitively expensive. Particularly for versioning systems that fragment file data.

For file systems that encrypt data on disk, data may be securely deleted by "throwing away" the corresponding encryption key [3]; without a key, data may never be decrypted and read again. This method works in systems that maintain an encryption key per file and do not share data between multiple files, unlike versioning systems and content-sharing stores [10]. This method greatly reduces the time needed to delete large amounts of data. The actual disposal of the encryption key often involves secure overwriting.

3 Secure Deletion with Versions

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 a secure data block (C_i), where $|B_i| = |C_i|$, and a short *stub* (s_i), whose length is a parameter of the scheme's security. In practice, s_i might be 128 bits. When the key (k) remains private, the transform acts as a secure authenticated encryption algorithm [2]. To securely delete an entire block, only the stub needs to be securely overwritten. This holds *even if the adversary is later given the key (k), e.g.* by subpoena. The stub reveals nothing about the data, and, thus, stubs may be stored on the same disk. A concept similar to stub deletion has been used in memory systems [**?**].

We present and compare two implementations of the keyed transform: one inspired by the all-or-nothing transform [4, 11], the other based on randomized keys. We also present extensions, based on key-sharing, that allow for the control and deletion of data by multiple parties.

3.1 AON Secure Deletion

The all-or-nothing (AON) transform [4, 11] ensures that an entire ciphertext, in our case a single file system block, must be decrypted before even one message block, some subset of the block, is revealed; no subset may be decrypted in isolation. The original intention of the AON transform was to increase the amount of time of a brute-force key search by a factor equal to the number of blocks in a ciphertext.

The all-or-nothing transform is the most natural construct for the secure deletion of versions. Our AON algorithm is presented in Figure 1(a). The algorithm takes a single file system block ($d_1, \ldots, d_m$), and performs encryption (Step 2) with a single file key, greatly easing key management. The encrypted data is authenticated (Step 3) and the result is then used to re-encrypt the data (Step 5). The resulting stub (Step 6) is not secret, rather, it is an expansion of the AON encrypted data.

AON encryption also enables the deletion of a block of data from an entire version chain. Due to the all-or-nothing properties of AON encryption, the

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

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

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

(a) Secure deletion using AON encryption

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

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

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

(b) Secure deletion using random key encryption

Figure 1: Two algorithms for authenticated encryption and secure deletion in versioning file systems.

secure overwriting of any 128 bits of a block will result in that block being securely deleted. This is the preferred technique for removing an entire version chain, as many blocks are shared between versions.

Despite these virtues, AON suffers from a known plain-text attack. After an encryption key has been revealed, if an attacker can guess the exact contents of a block of data, the attacker can verify that the data were once in the file system. This attack does not reveal encrypted data. Once the key is revealed, the attacker has all of the inputs to the encryption algorithm and may reproduce the ciphertext. The ciphertext may be compared to the undeleted block of data, minus the deleted stub, to prove the existence of the data.

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

3.2 Secure Deletion Based on Randomized Keys

To avoid such a plain text attack, systems must employ randomization, on a per-block basis, so that the encryption process is not repeatable. An algorithm for random-key secure deletion is shown in Figure 1(b). The scheme generates a random key, k, in Step 1 that is used to authenticate and encrypt a data block. To avoid the complexities of key distribution, we keep a single key per file, K, as with AON encryption, and use this key to encrypted the random key (Step 5). The encrypted randomly-generated key, c_0, serves as the stub. The expansion created by the AE scheme in Step 3 ($c_{m+1}, \ldots, c_n$) and the authentication of the encrypted random key (t) need not be securely overwritten to permanently destroy data. The encryption and storage of keys resembles lock-boxes in the Plutus file system [7].

The algorithm is built upon any Authenticated Encryption (AE) scheme (Step 3). This algorithm is provably secure when the underlying AE scheme is secure; AES and SHA-1 satisfy standard security definitions. An advantage of this transform is its speed. For example, when the underlying AE is OCB [12], only one pass over the data is made and it is fully parallelizable.

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

3.3 Secure Deletion with Secret-Sharing

Our random-key encryption scheme allows for the separation of the randomly-generated encryption key into key shares. Any number of randomly generated keys may be created in Step 1 (Figure 1(b)). and composed to create a single encryption key, k. With key shares, any single share may be destroyed to securely delete the corresponding data. However, all

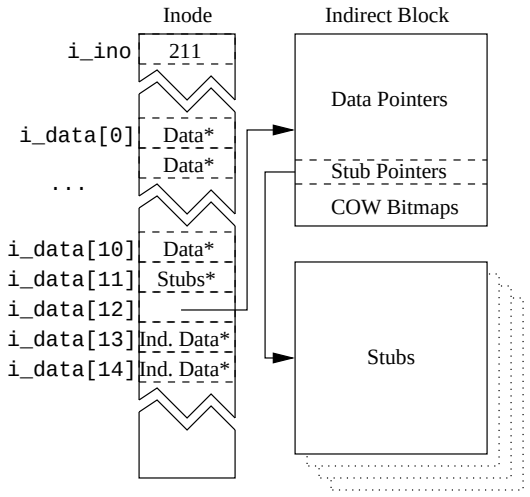

Figure 2: Metadata architecture to support stubs.

key shares must be present at the time of decryption. For example, a patient may hold a key share for their medical records on a smart-card, enabling them to control access to their records, and also independently destroy their records without access to the storage system.

4 Architecture

We are implementing secure deletion in ext3cow [9], 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 in which each inode represents a point-in-time snapshot of a file.

4.1 Metadata for Secure Deletion

Metadata in ext3cow have been retrofitted to support versioning and secure deletion. For versioning, ext3cow embeds bitmaps in its inodes and indirect blocks that record which blocks have had a copy-on-write performed. A 16-bit field is reserved in the inode itself to represent direct blocks. In a 4K indirect block (resp. doubly or triply indirect blocks), the last eight 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. Figure 2 illustrates our metadata architecture. The number of direct blocks in an inode has been reduced by one, from twelve to eleven, for storage of stubs that represent the direct blocks (`i_data[11]`). Ext3cow reserves additional 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 represent data in 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.

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 [15] will remove practical file size limitations.

4.2 Version Creation

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.

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 [14], 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%. 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.

Ext3cow will reduce the creation of versions based on the observation that multiple writes to the same stub may be aggregated in memory prior to reaching disk. Ext3cow will experiment with write-back caching policies that delay writes to stub blocks, looking to 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. Further, a small amount of non-volatile, erasable memory or an erasable journal would be helpful in delaying disk writes even when the system call specifies a synchronous write to disk.

5 Discussion

We recognize that there are many issues, beyond secure deletion, to securing versioning storage systems. For instance, securing the swap device and protecting against active memory attacks. While not comprehensive, secure deletion is an important technology for the regulatory environment.

We note that our secure deletion technology applies equally well to content-indexing systems. such as Venti [10] and LBFS [8]. Content-indexing stores a corpus of data blocks (for all files) and represents a file as an assemblage of blocks in the corpus. Thus, content-indexing shares blocks among files, as do versioning systems. Content indexing has the same deletion problems and our technology translates directly to this important, emerging research area.

We are in the midst of secure deletion research and development, and are currently implementing this scheme in ext3cow, available at *www.ext3cow.com*.

This work would not be possible without the hard work of Adam Stubblefield, Avi Rubin, and Randal Burns.

References

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

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

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

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

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

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

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

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

[9] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transcations on Storage*, 2005. To appear.

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

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

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

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

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

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