# Secure Data Deduplication

Mark W. Storer Kevin Greenan Darrell D. E. Long Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz
{mstorer,kmgreen,darrell,elm}@cs.ucsc.edu

## ABSTRACT

As the world moves to digital storage for archival purposes, there is an increasing demand for systems that can provide secure data storage in a cost-effective manner. By identifying common chunks of data both within and between files and storing them only once, deduplication can yield cost savings by increasing the utility of a given amount of storage. Unfortunately, deduplication exploits identical content, while encryption attempts to make all content appear random; the same content encrypted with two different keys results in very different ciphertext. Thus, combining the space efficiency of deduplication with the secrecy aspects of encryption is problematic.

We have developed a solution that provides both data security and space efficiency in single-server storage and distributed storage systems. Encryption keys are generated in a consistent manner from the chunk data; thus, identical chunks will *always* encrypt to the same ciphertext. Furthermore, the keys cannot be deduced from the encrypted chunk data. Since the information each user needs to access and decrypt the chunks that make up a file is encrypted using a key known only to the user, even a full compromise of the system cannot reveal which chunks are used by which users.

## Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; H.3 [**Information Systems**]: Information Storage and Retrieval

## General Terms

Design, Security

## Keywords

secure storage, encryption, cryptography, deduplication, capacity optimization, single-instance storage

## 1. INTRODUCTION

Businesses and consumers are becoming increasingly conscious of the value of secure, archival data storage. In the business arena, data preservation is often mandated by law [16, 26], and data mining has proven to be a boon in shaping business strategy. For individuals, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies and personal documents. Further, while few would argue that business data calls for security, privacy is equally important for individuals; data such as medical records and legal documents must be kept for long periods of time but must not be publicly accessible.

Paradoxically, the increasing value of archival data is driving the need for cost-efficient storage; inexpensive storage allows the preservation of all data that *might* eventually prove useful. To that end, deduplication, also known as single-instance storage, has been utilized as a method for maximizing the utility of a given amount of storage [4, 38, 5]. Deduplication identifies common sequences of bytes both within and between files ("chunks"), and only stores a single instance of each chunk regardless of the number of times it occurs. By doing so, deduplication can dramatically reduce the space needed to store a large data set.

Data security is another area of increasing importance in modern storage systems and, unfortunately, deduplication and encryption are, to a great extent, diametrically opposed to one another. Deduplication takes advantage of data similarity in order to achieve a reduction in storage space. In contrast, the goal of cryptography is to make ciphertext indistinguishable from theoretically random data. Thus, the goal of a secure deduplication system is to provide data security, against both inside and outside adversaries, without compromising the space efficiency achievable through single-instance storage techniques.

To this end, we present two approaches to secure deduplication: authenticated and anonymous. While the two models are similar, they each offer slightly different security properties. Both can be applied to single server storage as well as distributed storage. In the former, single server storage, clients interact with a single file server that stores both data and metadata. In the later, metadata is stored on an independent metadata server, and data is stored on a series of object-based storage devices (OSDs).

Both models of our secure deduplication strategy rely on a number of basic security techniques. First, we utilize convergent encryption [10] to enable encryption while still allowing deduplication on common chunks. Convergent encryption uses a function of the hash of the *plaintext* of a chunk as the encryption key: any client encrypting a given chunk will use the same key to do so, so identical plaintext values will encrypt to identical ciphertext values, regardless of who encrypts them. While this technique does leak knowledge that a particular ciphertext, and thus plaintext, already exists, an adversary with no knowledge of the plaintext cannot deduce the key from the encrypted chunk. Second, all data chunking and encryption occurs on the client; plaintext data is never trans-


*StorageSS'08,* October 31, 2008, Fairfax, Virginia, USA.

mitted, strengthening the system against both internal and external adversaries. Finally, the map that associates chunks to a given file is encrypted using a unique key, limiting the effect of a key compromise to a single file. Further, the keys are stored within the system in such a way that users only need to maintain a single private key regardless of the number of files to which they have access.

The remainder of this paper is organized as follows. In Section 2, we place our system within the context of the field's related work. Section 3 describes the threat model, which forms the basis of our design. In addition, Section 3 defines the assumptions, storage model, notation, and players in our secure, deduplication system. Section 4 provides a detailed description of how our system achieves improved storage utilization through deduplication, while providing data security. Section 5 provides an analytical examination of our system, including an evaluation of its security in a variety of scenarios. Finally, we conclude in Sections 6 and 7 with our future plans for this system and a short summary of our work.

## 2. RELATED WORK

Current systems that utilize single instance storage rely upon one of three primary deduplication strategies: whole file, fixed-sized chunks, and variable-sized chunks. The first, whole file, typically utilizes a file's hash value as its identifier. Thus, if two or more files hash to the same value, they are assumed to have identical contents and only stored once (not including redundant copies). This form of *content addressable storage* (CAS) is used in the EMC Centera system [14]. Farsite [10] and the Windows Single Instance Store [6] also perform deduplication on a per-file bases, though both use traditional identifiers and handle deduplication using a separate data structure. The second type of deduplication, per-block deduplication, is exemplified by the Venti archival storage system [27]. In Venti, files are broken into fixed sized blocks before deduplication, so files that share some identical contents (but not all), may still yield storage savings. The third, and most flexible form, breaks files into variable-length "chunks" using a hash value on a sliding window; by using techniques such as Rabin fingerprints [28], chunking can be done very efficiently. Variable-length chunks are used in LBFS [25], Shark [4], and Deep Store [38].

Many distributed file systems, such as OceanStore [29], SNAD [24], Plutus [20], and e-Vault [19], address file secrecy through the use of keyed encryption. The use of cryptographic techniques in these systems range from the assumption that all incoming data is already encrypted, to central architecture elements that define the system. However, none of these systems attempt to achieve the storage efficiency that is possible through deduplication. High-performance distributed file systems such as the Panasas Parallel File System [37], Ceph [35], and Lustre [7] typically have much less security than "standard" distributed file systems, trading higher performance for lower security. While there is an effort to add greater security to Ceph [21], this effort only involves authentication, not encryption.

At the opposite end of systems that provide secure, deduplicated storage efficiency, some systems utilize security models that incur a storage overhead. For example, PASIS [13] and POTSHARDS [33], achieve secure storage through secret sharing [31]. While well suited to long-term security, this technique incurs a very high storage overhead. Similarly, steganographic systems, such as the Steganographic File System [3] and Mnemosyne [15], provide plausible deniability over storage contents through the use of random data blocks. In both secret sharing and steganography, the storage overhead can be many times the size of the plaintext data.

In addition to data secrecy, several systems have addressed the demand for anonymity. Especially in the area of content distribution, there is a desire for systems that can hide the identity of data hosts, publishers and readers. For example, Publius uses encrypted data and secret sharing over keys to provide a censorship resistant web publishing platform that provides a high degree of writer anonymity [34]. Data encryption also provides the storage host with a level of plausible deniability; as there is no clear owner, a node's operator can claim that they have no knowledge about the plain-text data stored on their node [9].

Of all of these file systems, only Farsite combined deduplication with security. In its original design, its goal was to harness the unused disk space in a network of desktop-class computers, and present it as though it were a central file server [1]. In the original implementation, security was provided through file encryption where each user utilized a combination of symmetric and asymmetric keys. An extension of the work was an attempt to achieve better space efficiency through duplicate file coalescing [10]. To this end, the authors developed *convergent encryption*, in which the hash of the data is used as the encryption key. This allows users to independently encrypt identical plaintexts to the same ciphertext. However, unlike our work, Farsite only coalesces at the level of entire files. Our system coalesces data at a sub-file level, thus achieving space savings with files that are merely similar, as opposed to identical. Additionally, in the Farsite design, the client generates the encrypted value and its identifier. We show that if the key/value store for deduplicated data is not verified, its contents may be susceptible to targeted collision attacks. Finally, we present a model that allows secure, deduplicated storage in an anonymous user scenario.

## 3. THREAT MODEL

In order to properly design and evaluate a secure storage system, the threat model must be clearly laid out. In this section we identify the adversaries present in our model, our assumptions, and the attacks that must be considered in a secure deduplication system.

As part of establishing our security model, it is important to establish a consistent notation. There are two primary cryptographic functions that we utilize: encryption and hashing. An encryption function takes two parameters and is denoted $e(K,P) = C$, where $K$ is the encryption key, $P$ is the plaintext and $C$ is the ciphertext. An encryption function has a corresponding decryption function that uses a key and ciphertext to recover the original plaintext, expressed as $d(K,C) = P$.

Hashing is expressed in a manner similar to encryption. Simple hashing that does not utilize an encryption key is expressed as the single parameter function, $hash(P) = H$. Generating a hash of plain-text $P$ using a hash function and the encryption key $K_i$, known as an HMAC (keyed-Hash Message Authentication Code), provides integrity as well as message authentication. We express this function with the following notation: $HMAC_i(P)$.

### 3.1 Assumptions

One of the most fundamental assumptions that we make is that encrypted data is effectively random. The implication, in regards to deduplication, is that random data yields very low storage gains. We support this claim by examining the storage utilization of a system where each user encrypts data with their own distinct keys and the system deduplicates the encrypted data. For brevity, we assume the encrypted data is divided into fixed-sized chunks, though a similar argument can be made for variable-length chunks. At any point in time, the system is storing $k$ logical chunks of of length $l$. Each

chunk can take on any one of $2^l = m$ values. We recursively derive the physical storage utilization in chunks, $s_k$, as

$$
\begin{aligned}
s_1 &= 1 && (1)\\
s_k &= s_{k-1} + \left(1 - \frac{s_{k-1}}{m}\right) && (2)\\
&= s_{k-1}\left(1 - \frac{1}{m}\right) + 1 && (3)\\
&= \sum_{i=0}^{k-1}\left(1 - \frac{1}{m}\right)^i && (4)\\
&= m^{1-k}(m^k - (m-1)^k) && (5)
\end{aligned}
$$

If a single chunk exists in the system, then the number of logical and physical chunks is equal, thus our base case is $s_1 = 1$. Assuming a uniform distribution over encrypted chunks, the second chunk will match with probability $1/m$. In general, the $k$-th chunk matches with probability $s_{k-1}/m$. If $k \gg m$ then $s_k$ is very close to $m$, which results in a great deal of deduplication. Unfortunately, when $k \gg m$, performance may suffer and the size of file indices can become unwieldy. In practice, we find that $m \gg k$ and the utility of deduplication is extremely small because $s_k \approx k$. From this, we must conclude that deduplication of traditionally-encrypted data is largely ineffective. Since we are assuming that encrypted data is random, we can model deduplication of random data using the equation above. The chunk signature will operate on this random data and so should, if it is a good hashing function, be uniformly distributed as well. For chunks of non-trivial size—more than a few bytes—the likelihood of a match is extremely small and so, with extremely rare exceptions, every chunk will be unique and must be stored in its entirety.

Our second assumption is that encryption provides an adequate level of security for relatively short archival scenarios: if the data's lifetime is on the order of a few years, an attacker with access to ciphertext generated by a modern cryptosystem will be unable to determine the encryption key or derive the corresponding plaintext value. We recognize that in very long-term scenarios, on the order of decades, this assumption may not hold [32]. Extending this work into the secure, long-term area may be pursued as future work.

Next, we assume that an adversary that can sufficiently imitate a user has access to that user's data. In other words, if a malicious user has acquired enough information about the user—user names and passwords, for example—to participate in the system's protocols, then that user will obtain the standard results of that protocol. This scenario holds true in almost every secure system.

Because our solution utilizes hash functions in the generation of key material, we assume that they are cryptographically secure. More specifically, we assume that they are both weakly and strongly collision resistant. The former states that finding two input values that hash to the same output value is an intractable problem. The later, states that given a hash value, finding a value that hashes to the same output value is intractable.

Archival storage is typically used as a write-once, read-maybe store; thus, it stresses throughput rather than low-latency performance. Most existing large-scale deduplication systems are used as archival stores [27, 38]; the systems that are not often exhibit infrequent writes because they are well-suited for read-heavy workloads such as software distribution [4]. This usage pattern is quite different from the top storage tier of a hierarchical storage solution that stresses low-latency access and frequent writes, and also differs from backup solutions whose sole goal is high throughput writes, with reads an option of last resort. We assume that this emphasis on throughput allows the system to accommodate a reasonable latency penalty.

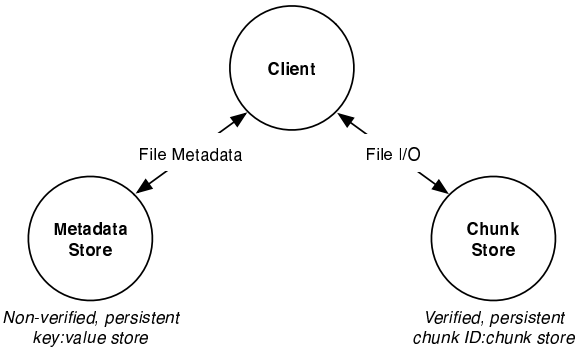


**Figure 1: The three primary players in the storage model and their interactions. In a distributed storage system, the responsibilities of the metadata server and chunk server are handled by separate clusters of systems. In a single-server model, the metadata and chunk server are on the same system.**

The data lifetimes we are considering are assumed to be on the order of years, not decades. While this is longer than the file lifetime often encountered in front-line storage [2, 22, 30], it is not as long as the indefinite lifetimes that other secure, archival systems are designed to support [33].

## 3.2 Players

As Figure 1 shows, at the protocol level, there are three primary players in our storage model: the client, metadata store, and chunk store. This arrangement maps to both single-server and distributed storage architectures. In a single-server architecture, the metadata store and chunk store are located on the same system, while a distributed storage system might choose to disconnect the metadata store and chunk server, handling the duties of each in separate clusters [35, 37].

Users interact with the system through the *client*, which is the starting point for both ingestion and extraction. As Figure 1 illustrates, it is the central contact point between the other components in the storage model. Unlike the other components in the storage model, the client does not have any persistent storage requirements, though the system assumes that users have reliable, secure access to their keys.

The *metadata store* is responsible for maintaining the information that users require in order to rebuild files from chunks — such as maps and encryption keys. We model this persistent storage using a simple, unverified key:value architecture. In such a system, when the user submits a key:value pair to the metadata server, the server does not need to verify that the key correctly corresponds to the value. For example, if the key is the hash of the value, the server does not need to verify that the hash of the value is the same as the key that the user submitted.

The role of the third player, the *chunk store*, is to persistently store data chunks, and to fulfill requests for chunks based on their ID. The chunk store is also modeled as a key:value store, however, unlike the metadata store, the chunk store must be able to verify the correctness of a the key with regards to the value. This is due to the possibility of targeted-collision attacks, as described below, that are possible within the chunk store.

In a deduplicated chunk store, a targeted-collision attack could be used to associate a false value with a given key. The pivotal difference between random collisions and targeted collisions is that a

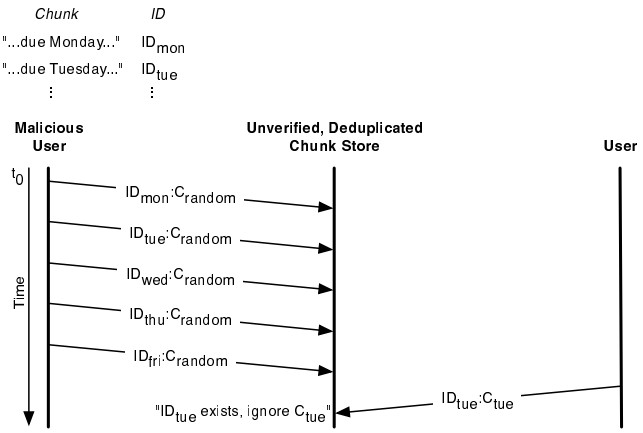


**Figure 2: Targeted-collision attack in which a malicious user exploits predictable data (in this example, a form letter with a due date) to generate valid chunk IDs, and associate those IDs with invalid chunks. If the user is the first to submit the ID, subsequent chunks will be deduplicated to a garbage value.**

user can exploit the predictable content of some data — in Fig 2 the malicious user utilizes similarities in form letters — to generate valid chunk identifiers. If an adversary can be the first to submit those identifiers with a garbage chunk, and if the chunk store cannot verify the correctness of the identifiers, subsequent submissions that have the same identifier will be deduplicated to the garbage chunk.

In addition to the three players of the storage model, our system identifies two adversaries, identified by their relationship to the system: external and internal. The external attacker exists outside of the system. This adversary does not have even simple insider access, such as a user account, and can only intercept messages or attempt to compromise a user's account.

In contrast, the internal attacker, or malicious insider, does have at least limited inside access. These attackers are further defined by their level of access, ranging from a simple user account access, to privileged root level access (as might be held by a malicious administrator). The existence of internal attackers implies that neither the metadata store, nor the chunk store are assumed to be trustworthy. Section 5 explores this implication by examining the security threat posed by internal adversaries.

The goal in both of the security models we present is to provide the users with a level of data protection from both external and internal attackers, regardless of the adversaries access level (or in the very least reduce the amount of data lost in a compromise). Each model provides an additional set of security features.

# 4. SYSTEM DESIGN

In this section we describe our two primary secure deduplication models: authenticated and anonymous. While similar, as Table 1 summarizes, each model offers a slightly feature set. We start by describing the security features, and then proceed to introduce the basic design of our secure deduplication techniques. Finally, we present the specifics of the two models. In particular we describe the contents of the metadata and chunk store for each, as well as their respective ingestion and extraction procedures.

The security property most associated with encryption is *secrecy*, which states that only authorized users are able to read plaintext data. Often, authorization is handled through key distribution; if a user is able to legitimately acquire the proper keys, she can access and decrypt the data. Both of the models that we present offer data secrecy against both external and internal adversaries.

| | Authenticated | Anonymous |
|---|---|---|
| Data secrecy | Yes | Yes |
| Anonymity | No | Yes |
| Per user revocation | Yes | No |
| Storage mode | Mutable | Immutable |

**Table 1: Security feature-set offered by our two secure deduplication models: anonymous, and authenticated. Note that anonymity is incompatible with per-user revocation.**

*Anonymity* allows the identity of a user to be hidden. This feature has two facets. The first is anonymity with respect to users submitting requests, *i. e.*, read and write requests cannot be attributed to a particular user. The second facet is anonymity with respect to storage contents, which states that the system is unable to determine which data is owned or accessible by a particular account.

*Revocation* is the ability to remove a user's access to a given file. In order for this to be done at a fine granularity, as in a per-user revocation, the system must include authentication; per-user revocation obviously cannot exist in a system without knowledge of a user's identity. Revocation schemes can be described by the action that takes place at the time of the revocation. In active revocation, access is immediately removed. This is often expensive, and may involve a fair amount of cryptographic computation. In lazy revocation, access is only removed when the data is changed. Thus, the user is unable to see any changes that occur after the revocation, but may have continuing access to what they were previously entitled to view.

## 4.1 Secure Deduplication Overview

In both the anonymous and authenticated models, clients begin the ingestion process by transforming a file into a set of chunks. This is often accomplished using a content-based chunking procedure which produces chunks based on the contents of the file. The advantage of this approach is that it can match shared content across files even if that content does not exist at the multiple of a given, fixed offset [25]. The algorithm selects chunks based on a threshold value $A$ and a sliding window of width $w$ that is moved over the file. At each position $k$ in the file, a fingerprint, $F_{k,k+w-1}$, of the window's contents is calculated [28]. If $F_{k,k+w-1} > A$, then $k$ is selected as a chunk boundary. The result is a set of variable sized chunks, where the boundary between chunks is based on the content of the data.

Both file chunking and encryption occur on the client. There are a number of benefits to performing these tasks on the client, as opposed to the server. First, it reduces the amount of processing that must occur on the server. Second, by encrypting chunks on the client, data is never sent in the clear, reducing the effectiveness of many passive, external attacks. Third, a privileged, malicious insider would not have access to the data's plaintext because the server does not need to hold the encryption keys.

Clients encrypt chunks using *convergent encryption*, which was introduced in the Farsite system [10]. Using this approach, clients use an encryption key deterministically derived from the plaintext content to be encrypted; both Farsite and our system use a cryptographic hash of the plaintext as the key. Since identical plaintexts result in the use of identical keys, regardless of who does the encryption, a given plaintext always results in the same ciphertext.

$$K = hash(\text{chunk}) \tag{6}$$

Compared to other approaches, this strategy offers a number of advantages. As we have shown in Section 3, if each user encrypted using his own key, the amount of storage space saved through deduplication would be greatly reduced because the same chunk encrypted using two different keys would be would result in different ciphertext (with very high probability). Second, attempting to share a random key across several user accounts introduces a key sharing problem. Third, a user that does not know the data plaintext value cannot generate the key, and therefore cannot obtain the plaintext from the ciphertext. This point is especially important since, in contrast to an approach where the server encrypts the data, even a root level administrator does not have access to a chunk's plaintext value without the key.

The primary security disadvantage of this approach, as identified in its original description [10], is that it leaks some information. In particular, convergent encryption reveals if two ciphertext strings decrypt to the same plaintext value. However, this behavior is necessary in systems that use deduplication, since it allows a system to remove duplicate plaintext data chunks while only observing the ciphertext; information leakage is part of the compromise needed to achieve space-efficiency through deduplication.

Each ciphertext chunk must be assigned an identifier. In our system, each chunk in the system is identified using the encrypted chunk's hash value, a technique sometimes referred to as content-based naming.

$$\text{chunk\_id} = hash(e(hash(\text{chunk}), \text{chunk})) \tag{7}$$

An alternative to using the hash of the encrypted chunk is to use the hash of the hash of the plain-text chunk, *i. e.*, the hash of the encryption key is the chunk identifier. This approach offers a number of attractive qualities. First, performance is improved. In both approaches the user performs two hashes: a key generation hash, and an identifier generation hash. Assuming that key lengths are smaller than chunk lengths, performing two chunk hashes will be more expensive than a chunk hash and a key hash. Second, if the identifier can be derived from the key, then the file to chunk map only needs to preserve the key, as opposed to the key and the identifier. However, there is a large drawback of using the hash of the key as the identifier: the chunk store cannot verify that the chunk's content-based identifier is correct. As Section 3.2 explained, unverified chunk signatures permit the use of targeted collision attacks.

The encrypted chunks themselves are stored within the chunk store. In a distributed storage model, where there may be multiple chunk stores, the chunk list can also include the information needed to locate the correct storage device. Alternatively, deterministic placement algorithms can be used to locate the correct storage devices based on the chunk's identifier [18, 36, 8].

## 4.2 Authenticated Model

The authenticated model is the most similar to the original design of convergent encryption as it is utilized in Farsite [10]. As with their design, our authenticated model makes a number of assumptions regarding encryption keys and the key management techniques available to users. First, we assume that each user has a symmetric key that is private to that user. Second, we assume that each user also has an asymmetric key pair. Third, it is also assumed that a certificate authority exists to facilitate the trusted distribution of public keys. Finally, it is assumed that users are able to generate cryptographically sound encryption keys.

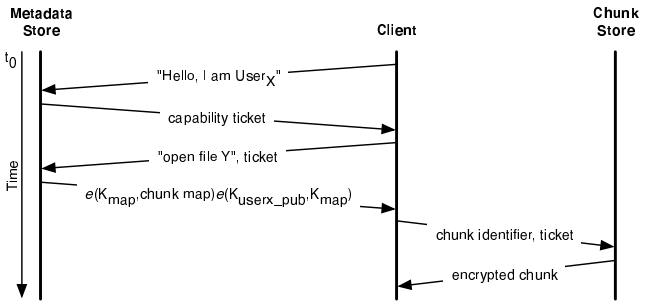


**Figure 3: Extraction in the authenticated model begins with the client contacting the metadata store for the secure chunk map, and the chunk map's encryption key. From there, subsequent communication involves requesting chunks from the chunk store.**

Ingestion begins with the client identifying the chunks and then encrypting them using convergent encryption. Following this, the information needed to rebuild the files, including chunk locations, names and encryption keys, is stored within a chunk map. As Table 2 illustrates, this map is stored in the metadata store in a map entry and accessed through a file's inode number. Additionally, it is encrypted using a dedicated map key. To allow authorized users to decrypt the map, the map key is encrypted using the authorized users' public keys. These encrypted keys are identified via a user identifier, and appended to the end of the encrypted chunk map; this technique is similar to the widely used "lockbox" approach to encrypting files [24, 20]. As more users are granted access to the file, additional encrypted keys can be appended to the map entry. The final step of ingestion is to submit the encrypted chunks to the chunk store. As Subsection 3.2 discussed, the chunk store is capable of generating the chunk IDs, so the client is not required to submit an identifier along with each chunk.

Extraction, as Figure 3 illustrates, follows a communications path similar to that of ingestion. The process starts with the client authenticating to the metadata store and submitting an `open()` request. As shown in Table 2, the metadata store can use the file's inode number to locate the encrypted chunk map and list of encrypted map keys. Rather than return the chunk map and the entire list of keys, the metadata store will only return the chunk map and the key that corresponds to the user, resulting in less information to transmit, and not leaking to the user the list of all users that have access to the file. Finally, the client decrypts the map key and subsequently the chunk map, and directs chunk requests to the appropriate chunk store.

In addition to allowing multiple users access to a single chunk map, the list of encrypted map keys also plays a central role in revocation. If access to a file needs to be revoked for a specific user, a new chunk key can be generated, the chunk map is encrypted using the new key, and the list of encrypted keys is updated for the users that still have access.

## 4.3 Anonymous Model

The goal of the anonymous model is to hide the identities of both authors and readers. This model operates under the assumption that encrypted data is secure against an adversary that does not posses the correct encryption key; thus, authentication is unnecessary.

One of the drawbacks of an anonymous data-store is that both well-behaved and malicious users are anonymous. This opens the door to attacks in which authorized users perform malicious acts,

| Metadata Store | Key | Value |
|---|---|---|
| File inode | file name | inode number |
| Map entry | inode | $e(K_{map}, \text{chunk map})[(\text{uid}, e(K_{user\_pub}, K_{map}))]$ |

| Chunk Store | Chunk ID | Encrypted Chunk |
|---|---|---|
| | $hash(\text{encrypted chunk})$ | $e(hash(\text{chunk}), \text{chunk})$ |

**Table 2: Authenticated model persistent storage details. The map entry stores a chunk map (an ordered list of the data needed to request and decrypt chunks) and is encrypted using a dedicated map key. This key is then encrypted using the public key of users that are authorized to access the file and appended to the encrypted chunk map.**

| Metadata Store | Key | Value |
|---|---|---|
| File entry | file name | inode number |
| Map reference | $HMAC_{user}(\text{inode})$ | $e(K_{user}, K_{map})$ |
| Map entry$_i$ | $HMAC_{map}(\text{inode}, \text{map}_{i-1})$ | $e(K_{map}, [\text{chunk map}])$ |

| Chunk Store | Chunk ID | Encrypted Chunk |
|---|---|---|
| | $hash(\text{encrypted chunk})$ | $e(hash(\text{chunk}), \text{chunk})$ |

**Table 3: Details of the anonymous model for a persistent chunk store. The model makes exclusive use of symmetric keys. Map references are used to store the map encryption key in a manner that allows users to see changes made by other authorized members.**

such as deleting or changing data, under the assumption that they cannot be definitively identified. One of the ways to guard against such an attack is to make both the metadata and chunk stores immutable. This approach thus implies that file changes are reflected in a versioned history of chunk maps, similar to the mechanism used in WAFL to transition to a new version of the file system [17]. Thus, as in systems such as SUNDR [23], malicious changes are isolated in a branch of the original file.

As Table 3 illustrates, the system makes exclusive use of symmetric keys; thus, the model must make several assumptions regarding encryption keys. First, it is assumed that each user has a symmetric encryption key that is private to the user. The use of a symmetric key for each user does not compromise anonymity because the key is used in HMAC procedures and, therefore, only the owner can confirm that the hash was created with their key. Second, it is assumed that users have the ability to generate cryptographically sound keys. Third, it is assumed that users are able to communicate off-line (relative to the chunk store) for the purposes of sharing keys.

In the anonymous model, as in the authenticated model, ingestion begins with the client identifying and encrypting chunks. The information needed to reconstruct the file from chunks is written to a map entry and encrypted using a map key that is generated when the file is created. This map entry is then written to the metadata store.

As Figure 4 shows, the system utilizes a map reference, specific to each user, that holds the map key to allow multiple users to see file changes made by other authorized users. Thus, file access is granted outside of the system by sharing the map key. When the user is given a map key, they create and store a map reference in the metadata store. In this manner, the only key the client is required to remember is their private symmetric key.

If changes are made to a file, the new map entry is written to the metadata store as a linked list. As Figure 4 illustrates, each user's map reference is used to locate the root of this linked list through an HMAC keyed with the map key. Each time a client commits a write

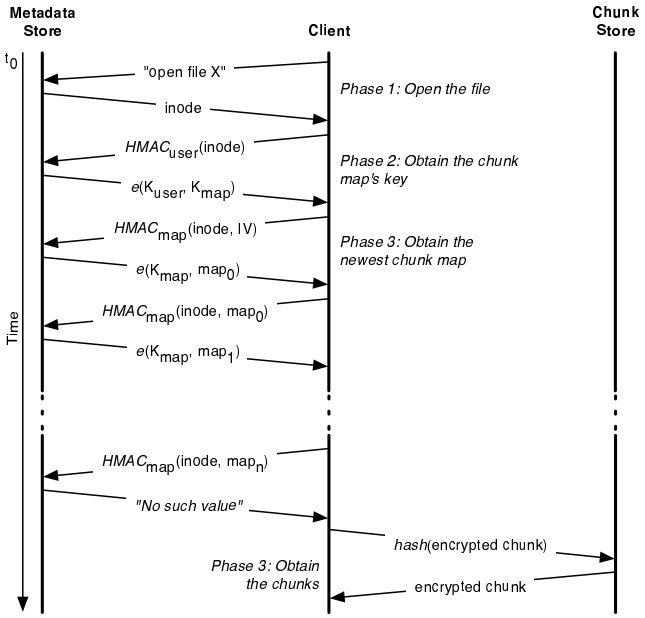


**Figure 5: Extraction in the anonymous model begins with an `open()` request which returns an inode number. Using the file's inode number and the user's symmetric key, a user can obtain the chunk list through his chunk list reference. Requests for chunks are then directed to the chunk store.**

to the system, a new node is appended to the list. Traversal of the list is accomplished through an HMAC, keyed with the map key, of the inode, and the previous map entry. As with the authenticated model, the client submits chunks to the chunk store in the final stage of ingestion.

Sharing files could also be accomplished by using the authorized user's symmetric key to encrypt the map key, and appending this encrypted key to the chunk map. While similar to the authenticated model's strategy, this approach suffers from a number of disadvantages. First, any information that identifies the user's key in the list is breaking anonymity. Second, even if an HMAC of the inode was used to hide the user's identity, the list would still leak the number of users that have access to the file. Third, the use of map references provides a level of coarse grained revocation. A chunk map can be created and encrypted with a new chunk map key. This new key would then need to be distributed to authorized users, who in turn can create new map references.

File extraction in the anonymous model, illustrated in Figure 5, proceeds in four phases. First, the client contacts the metadata store to issue an `open()` request and obtain the file's inode. Second,

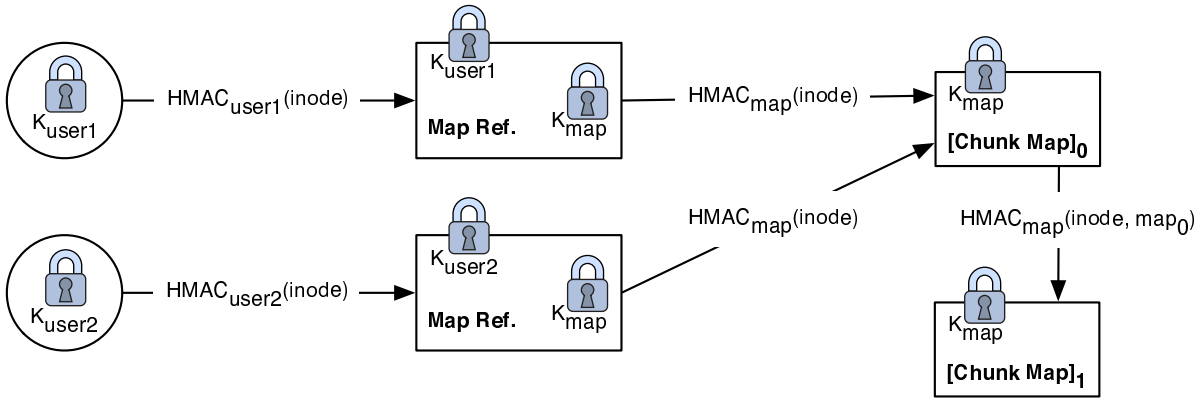


**Figure 4: The information needed to reconstruct files is stored as a linked list of immutable chunk maps that are encrypted using a dedicated key, $K_{map}$. Each user creates a map reference, protected by their unique symmetric key, to store the map key. This allows users to see changes created by other authorized users, while only requiring them to remember one key (their unique user key).**

the client obtains the map's key by utilizing the map reference, as shown in Figure 4. In the third phase, the client traverses the linked list of map entries by issuing map requests until they arrive at the version they want, or the map request fails, indicating that the end of the list has ben reached. In the fourth and final phase, the client utilizes the map entry and map key to determine which chunks to request from the chunk store.

# 5. SECURITY ANALYSIS

The evaluation of the two secure deduplication models that we have presented is intended to demonstrate that the system is secure in the face of a variety of foreseeable scenarios. First, we examine the attacks that an external adversary could inflict upon the system. Second, we examine the security leaks possible when faced with a malicious insider who might have access to all of the raw data, such as system administrator with root-level access. Third, we examine the security implications involved when the keys in the system become compromised.

## 5.1 External Adversaries

For a system to be considered secure, it must be able to prevent information from leaking to an external attacker. A passive example of such an adversary would be an attacker that intercepts messages sent between players in the system. An active example is an adversary that changes or transmits messages.

In both the authenticated and external model, the passive attacker problem is largely ameliorated by having the client perform the chunking and encryption. Thus, plaintext data is never transmitted in the clear. However, the anonymous model assumes that the keys can be exchanged in a secure manner but does not explicitly state how this is accomplished. A potential area of future work could be to define a secure protocol for this procedure.

Since data transmitted between players is always encrypted, the danger from an active adversary is one of messages being changed. For example, in the basic models we have presented, a chunk could be intercepted en route to the chunk store and modified. While our design does not explicitly address such scenarios, these attacks can be largely mitigated through the use of transport layer security (TLS) approaches such as Secure Sockets Layer.

As the anonymous model includes the goal of hiding the user's identity, an external adversary can gain some information by identifying where requests originate from. As with the man-in-the-middle type attacks previously discussed, our system does not directly deal with the issue, however solutions such as onion routing have addressed this concern, and are compatible with our design [12].

## 5.2 Internal Adversaries

As discussed in Section 3, a secure system must also provide protection from internal attackers. To this end, we analyze the ability of an inside adversary to launch attacks based on their location within the system and across their potential access levels.

As in most systems, a malicious insider with full access can change or delete any information he chooses, resulting in a denial of service attack. From a security standpoint, our goal is, therefore, to limit an insider's ability to make targeted changes. There are two facets to limiting such changes. First, we would like to limit an insider's ability to target specific files. Second, we would like to limit an adversary's ability to make undetectable changes; overwriting a value with garbage is generally more detectable that overwriting it with a semantically valid, but incorrect value.

### *5.2.1 Authenticated Model*

In the authenticated model, the metadata server does leak some information to an internal adversary. First, an insider has access to the file name to inode mapping. Second, the inode number to encrypted map entry is also available to an internal adversary. Finally, a malicious insider can determine the files to which a user has access, and the users that have access to a specific file.

Using the information available, a inside attacker at the metadata server is able to launch a variety of attacks. First, an inside adversary can delete metadata and revoke access for specific users. If the client is not knowledgeable about which files it should be able to access, this attack is undetectable. Second, when a client requests a file, the map entry of a different file accessible by the client could be returned. Whether or not this attack is detected would rely upon the client's understanding of the file's contents.

Targeted changes to file contents, however, require the adversary to obtain the map key. In the current design, users grant access by submitting map keys encrypted using the authorized user's public key. In this way, a malicious insider is never exposed to the plaintext key needed to access a map entry's details. If the system were to encrypt map keys, a malicious insider could change the contents of map entries. One way to further strengthen the system, then, would be to hide the map entry from an inside attacker. This could be accomplished using a technique such as the anonymous model's

map references, which, as shown in Figure 4, requires the map key in order to locate the map entry.

Finally, if a malicious insider at the metadata store also distributes capability tickets, as is done in some systems, then it can be assumed that the adversary also has access to chunks; a malicious metadata store can simply issue itself a valid capability. However, without access to the map key, the adversary would not know which chunks correspond to a give file, and would lack the key needed to decrypt a chunk.

#### *5.2.2 Anonymous Model*

In both the authenticated and anonymous model, an inside adversary at the chunk store would be unable to modify data without being detected. Since the name of the chunk is based on the content, a user would not be able to request the modified chunk, or at the very least could tell that the chunk they requested is different from the chunk that was returned to them. An insider at the chunk store could, of course, delete chunks or refuse to fulfill chunk requests.

In the anonymous model, the metadata store does leak some information to an internal adversary. First, an insider can deduce which inode numbers map to which files. This is not a serious issue because the user's symmetric key is needed to map inodes to map references. More importantly, however, an insider could deduce which entries are map references, as they will all be the same length. This is due to the fact that their payload is always one key, as opposed to a variable list of chunk metadata. One way to avoid leaking the fact that an entry is a map-key is to append some amount of random data to the entry.

### 5.3 Key Compromises

Any system that utilizes cryptographic primitives is highly dependent on the controlled access of encryption keys for the security of the system. As Kerckhoff's principle states, the security of the system comes from an adversary not knowing the encryption key; it is assumed that the adversary knows the protocols and cryptosystems. Thus, one way to analyze a security system is to examine the effects of compromised keys.

#### *5.3.1 Authenticated Model*

In the authenticated model, the user's identity is tied to their asymmetric key pair. Further, if an adversary learns a users private key, it is assumed they have the users complete key pair; the public key can easily be acquired from a certificate server. In this scenario, a malicious user may be able to fully impersonate the key's rightful owner, and obtain all the abilities of that user. As a safeguard against this possibility, it is recommended that authentication require more information than the user's key, but this approach is outside the scope of our model.

A compromise of the other metadata key in the authenticated model, the map key, results in a less drastic information leak. If an adversary learns the map key, the problem of authenticating to the metadata store still exists. Finally, the revocation process can be used to generate a new map key, making the old key invalid. Thus, the system is relatively safe in the event of a compromised map key.

Similarly, if the last key of the authenticated model, the chunk key, is compromised, the information leak is rather small. This is due to the fact that an adversary with the chunk key would still need to know the chunk identifier, and be able to authenticate to the chunk server in order to obtain plaintext data.

#### *5.3.2 Anonymous Model*

In the anonymous model, the user's private, symmetric key is very important to the security of the system. If a malicious user obtains the user's key, it can be safely assumed that they can access any file that the user has stored a map reference for. Another potential attack they can issue in this scenario is to extend the length of the linked list of map entries indefinitely. However, since the anonymous model uses immutable chunks, a new key could be generated, and the file branched.

If an adversary obtains the map key, the adversary will only need the inode number of a the file to obtain plaintext data. Assuming that the number of inodes is relatively small, this can be accomplished using a brute force attack. Additionally, as the system is immutable, even generating a new map key will result in the original file being compromised.

As in the authenticated model, an adversary with the chunk's encryption key, would still need to know the chunk identifier in order to obtain plaintext data.

## 6. FUTURE WORK

While the models we have presented demonstrate some of the ways that security and deduplication can coexist, works remains to create a fully realized, secure, space efficient storage system. Open areas for exploration exist in both security, as well as deduplication.

Storage efficiency can be increased in a number of ways through intelligent chunking procedures. For example, the size of the file may be used to determine the average chunk size, potentially yielding greater deduplication in data such as media files, which tend to be large and exhibit an "all or nothing" level of similarity with other files. However, since some large files, such as mail archives or `tar` files, may be aggregations of smaller files, another possibility would be to adjust chunking parameters based on file types. Since chunking is done at the clients rather than at the servers, this approach only requires that clients agree on the way they divide files into chunks. Moreover, taking this approach does not increase the likelihood of collision, which remains very small for chunk identifiers of 160 bits or longer.

Unfortunately, techniques such as delta compression on files or chunks [11, 38], while they have proven effective for standard deduplicated storage systems, may not work well with encryption because clients cannot access encrypted chunks for any files but their own, limiting the source material for deduplication. Moreover, approaches that try to locate similar chunks in a chunk store will likely be ineffective because they require plaintext data for indexing; similar, but not identical, plaintext chunks will result in ciphertext chunks that have no similarity.

Another way to increase storage efficiency would be to provide deletion and garbage collection. While these are straightforward to implement in many systems, storage reclamation can be difficult in a system that uses deduplication because a single chunk may be referenced by many different files. Thus, removing a chunk requires an understanding of how many files reference the chunk. A common approach to deletion in a deduplicated file system uses reference counts to track the number of files that use a particular chunk. Of course, such a system would need to ensure that a malicious user could not launch a denial of service type attack simply by deleting chunks or modifying reference counts.

Currently, our model provides an all or nothing level of access; if a user has the map key, they have access to the file. Future designs could utilize multiple levels of permissions. Thus, a user could be allowed to read a file, but the system would prevent them from deleting information.

Finally, even in the anonymous model, a secure capability from the metadata store could be used to implement file locking. Currently, there is no guard against multiple users writing to the same chunk map. While the anonymous model is immutable and therefore all versions of a file are present, locking would still allow for changes to appear at the correct order in the list of maps. This would, of course compromise some of the users anonymity, as their requests would form a distinct session.

# 7. CONCLUSION

We have developed two models for secure deduplicated storage: authenticated and anonymous. These two designs demonstrate that security can be combined with deduplication in a way that provides a diverse range of security characteristics. In the models we present, security is provided through the use of convergent encryption. This technique, first introduced in the context of the Farsite system [1, 10], provides a deterministic way of generating an encryption key, such that two different users can encrypt data to the same ciphertext. In both the authenticated and anonymous models, a map is created for each file that describes how to reconstruct a file from chunks. This file is itself encrypted using a unique key. In the authenticated model, sharing of this key is managed through the use of asymmetric key pairs. In the anonymous model, storage is immutable, and file sharing is conducted by sharing the map key offline and creating a map reference for each authorized user.

In our evaluation, we have analyzed the security of each model with regard to a number of security compromises. We found that the system is mostly secure against external attackers. Further, the security threats that our models do not explicitly guard against can be addressed through the addition of standard secure communications techniques such as transport later security. Security compromises by a malicious insider are largely mitigated from the design's avoidance of server side encryption. Since insiders are never exposed to plain-text or encrypted keys, their ability to change metadata values in an undetectable way is greatly diminished. Security is even more apparent in the chunk store where the content addressed nature of secure chunks intrinsically makes the detection of malicious changes quite noticeable. Finally, we examined the information leaks resulting from key compromises and found that the most severe security breaches result from the loss of the client's key. The damage in the event of such a key loss is confined, however, to the user's files. Moreover, the breach of client's keys is a serious threat in most secure systems.

## Acknowledgments

We thank the members of the Storage Systems Research Center (SSRC) for spirited discussions that helped focus the content of this paper. This work was supported in part by the Department of Energy under award DE-FC02-06ER25768 and industrial sponsors of the Storage Systems Research Center at UC Santa Cruz, including Agami Systems, Data Domain, Hitachi, LSI Logic, NetApp, Seagate, and Symantec.

# 8. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, Feb. 2007.

[3] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Proceedings of the International Workshop on Information Hiding (IWIH 1998)*, pages 73–82, Portland, OR, Apr. 1998.

[4] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, 2005.

[5] D. Bhagwat, K. Pollack, D. D. E. Long, E. L. Miller, J.-F. Pâris, and T. Schwarz, S. J. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of the 14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)*, Monterey, CA, Sept. 2006.

[6] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. USENIX, Aug. 2000.

[7] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[8] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler. Dynamic and redundant data placement. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, 2007.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–66, 2001.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 617–624, Vienna, Austria, July 2002.

[11] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126. USENIX, June 2003.

[12] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 1999.

[13] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 Int'l Conference on Dependable Systems and Networking (DSN 2004)*, June 2004.

[14] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd Int'l Symposium on Computer Architecture*, pages 60–71, June 2005.

[15] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. *Lecture Notes in Computer Science*, 2429:130–140, Mar. 2002.

[16] Health Information Portability and Accountability Act, Oct. 1996.

[17] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994*

*USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

[18] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[19] A. Iyengar, R. Cahn, J. A. Garay, and C. Jutla. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC '98)*, pages 123–135, Sept. 1998.

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

[21] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*, Nov. 2007.

[22] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[23] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[24] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

[25] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Oct. 2001.

[26] M. G. Oxley. (H.R.3763) Sarbanes-Oxley Act of 2002, Feb. 2002.

[27] 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, Monterey, California, USA, 2002. USENIX.

[28] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[29] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Mar. 2003.

[30] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

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

[32] M. W. Storer, K. M. Greenan, and E. L. Miller. Long-term threats to secure archives. In *Proceedings of the 2006 ACM Workshop on Storage Security and Survivability*, Alexandria, VA, Oct. 2006.

[33] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS: secure long-term storage without encryption. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 143–156, June 2007.

[34] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, pages 59–72, Denver, CO, Aug. 2000.

[35] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006. USENIX.

[36] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, Nov. 2006. ACM.

[37] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, Feb. 2008.

[38] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.