Trustworthy Migration and Retrieval of Regulatory Compliant Records

Soumyadeb Mitra Marianne Winslett
University of Illinois at Urbana Champaign
$\{$*mitra1,winslett*$\}$*@cs.uiuc.edu*

Windsor H. Hsu Xiaonan Ma
IBM Almaden Research Center
$\{$*windsor,xiaonan*$\}$*@almaden.ibm.com*

Abstract

Compliance storage servers are designed to meet organizational needs for trustworthy records retention, largely mandated by recent legislations such as HIPAA, SEC Rule 17a, and the Sarbanes-Oxley Act. These devices export a file-system-level interface, and enforce write-once read-many (WORM) semantics for file access. Compliance storage protects records from alteration, as long as they remain on the same storage server. However, the decades-long records retention requirements of recent legislation mean that a compliance storage server will often be obsolete long before the documents it contains can be destroyed. Unfortunately, records will be vulnerable to change during migration to a new server. Records are also vulnerable during retrieval, when they are taken off the server and "migrated" to the person or organization who needs them. In this paper, we propose techniques for trustworthy document migration and retrieval, by enhancing the storage servers with the capability to sign their files and directories. The proposed techniques can be used to verify that a migration was carried out properly, even across multiple migrations, deletions of expired documents, and changes in the content and structure of migrated directories. In our approach, file writers incur no performance penalty, which is important since compliance workloads are write-intensive. Migration incurs a reasonable 5-10% space overhead and requires 24 msec processing time per file. The result of the migration can be verified at a rate of 24 msec per file by a trustworthy auditor (or ordinary user), who can then generate a certificate attesting to the correctness of the migration.

1. Introduction

The trustworthy retention of financial information, customer communications, medical images, drug development logs, quality assurance test reports, and other important documents is increasingly being mandated by government regulations, such as the US Sarbanes-Oxley Act [8] and SEC Rule 17a-4 [28]. At each point in their life cycle, these documents must be readily accessible. Non-compliance with these regulations has recently resulted in stiff penalties from the Securities Exchange Commission [2], with accompanying bad publicity and potential investor flight.

Organizations have been addressing these needs by purchasing *compliance storage servers* [11, 15, 17, 23]. These are file servers that store documents on WORM storage, implemented by ordinary magnetic disks surrounded by a file or object interface that prevents users from overwriting previously-written bytes. Additional features prevent users from removing, tampering with, or replacing the server's disks [16]. Unfortunately, merely using WORM storage, as is the current focus, is insufficient to ensure that documents are trustworthy—the entire document life cycle must be secured, from the moment of creation on through storage, maintenance and retrieval. This life cycle may span decades. For example, HIPAA requires medical records of infants to be retained for 21 years. The Occupational Safety and Health Administration (OSHA) requires employers to keep records on exposure of employees to toxic substances for 30 years.

It is unlikely that a record can be stored on a single server for such long periods of time. Many practical considerations will mandate the migration of records from one storage server to another, such as the challenges of maintaining obsolete storage servers (no vendor support, archaic underlying supporting technology, and the cost and difficulty of finding support staff), corporate mergers and spin-offs, the introduction of storage servers with much larger capacity, the need to migrate data from a backup server in the event of device failure, and so on. However, migration of records introduces a weak link in the process of ensuring trustworthiness. A powerful insider adversary such as a CEO can modify or remove records while they are being moved from one device to another. The goal of trustworthy migration is to ensure that any such attempt to tamper with the records during migration can be quickly and easily detected.

In this paper, we propose techniques for supporting trustworthy policy-driven migration of compliance records to new compliance storage servers. Every migration involves three steps. First the (untrusted) migrator creates a log of planned operations. The log defines the file and directory omissions and restructurings that will take place during migration. Then the migrator has the current storage server generate certificates attesting to the current contents of its directory tree. Finally, the migrator puts the migrated files, directories, and logs on the new storage server.

After migration, an auditor (or an ordinary user) can run validation routines to determine that the files and directories that they access have been migrated correctly from the servers where they were originally committed. Subsequent readers can trust the migration based on the certificate of migration generated by the trustworthy auditor, or can perform their own validation if desired.

The remainder of this paper is organized as follows. We present the storage model, threat model, and supported functionality in Section 2. Related work is discussed in Section 3. We propose the architecture for trustworthy migration in Section 4. In Section 5, we show how the architecture can be used to support trustworthy migration, including special cases such as migration into a nonempty device or multiple migrations. Section 6 discusses techniques to support policy-driven migration. A performance evaluation appears in Section 7. Finally, we conclude in Section 8.

2. Background

Storage Model In this paper, each record is a single file that has been written to a compliance storage server. The compliance storage server offers an ordinary file system interface (with a few restrictions described below) to users over a remote file access protocol. While NFS is popular for this purpose today, new protocols may be adopted over the decades. We allow the file system interface to change, as long as files are still organized in a directory hierarchy and metadata regarding the name, size, owner and expiry date is kept for each file and directory. (Support for additional metadata is a straightforward extension.) Although we focus on a storage server with a file system interface, our ideas can be extended to object servers.

We assume that the file system interface will allow users to create new directories and files and to append to existing regular (i.e., non-directory) files; it will not allow users to overwrite previously written file bytes. (We include append operations because they are required for trustworthy indexing [20].)

At its creation, each file and directory is given an expiry time by its creator or through a previously declared system of default expiry times. Once assigned, a file or directory expiry time cannot be moved earlier, but can be moved later. The file system prevents the deletion of existing files and directories before they expire. Once a file or directory expires, the storage server can delete it, on instruction from the user or an application. If a directory is deleted, all the files and directories under it should also be deleted. The expiry time of a directory should hence be later than that of all the files and subdirectories under it.

Threat Model The primary non-physical threat to compliance records is *undetectable alteration or destruction of existing records at the behest of high-ranking company insiders* such as CEOs and CFOs, who wish to retroactively hide activities documented in the organization's compliance records. For example, Ralph may want to hide an email conversation that he had with Martha about his expectations of a drop in the stock price for his company. Because the threat comes from high-ranking insiders, attackers can have superuser privileges, can initiate and control the migration process, and can attempt to modify or omit records during migration.

More formally, a legitimate user, say Alice, creates a file *R* and commits it to the compliance server. We trust the document insertion code used by Alice to create *R*, so *R* does reach compliance storage initially. Moments or decades after *R* has been created, a malicious user Mala starts regretting its existence. Mala wants to prevent every future legitimate user Bob, who may be a prosecutor or auditor, from retrieving *R*.

We assume that the compliance server itself is trustworthy, i.e., no bytes of user files are ever rewritten, and every requested read/write/append operation is performed correctly. Our migration scheme relies on the ability of the storage server to sign files and directories using a public key cryptosystem. The server private key must be properly secured so that it is unavailable to the adversary (we propose a storage architecture for achieving this in the longer version of our paper [21]). Bob will have to learn the names and public keys of all the storage servers that have ever been used in the company. The names and public keys themselves can be certified by a trusted third party such as the storage server vendor. This also prevents the adversary from launching a man-in-the-middle attack by introducing fake compliance storage servers (for which he knows the private keys) and migrating data through them.

Because the compliance server enforces the WORM semantics properly, Mala can modify or omit records only during migration. A successful attack on migration consists of *undetected* tampering. If tampering is detected, an investigation (with presumption of guilt) will be launched immediately. Thus we aim for a tamper-evident migration system. For the same reason we are not concerned with physical destruction or loss of compliance servers.

Migration Scenario *R* can be migrated from one compliance server to another (server-to-server migration), or

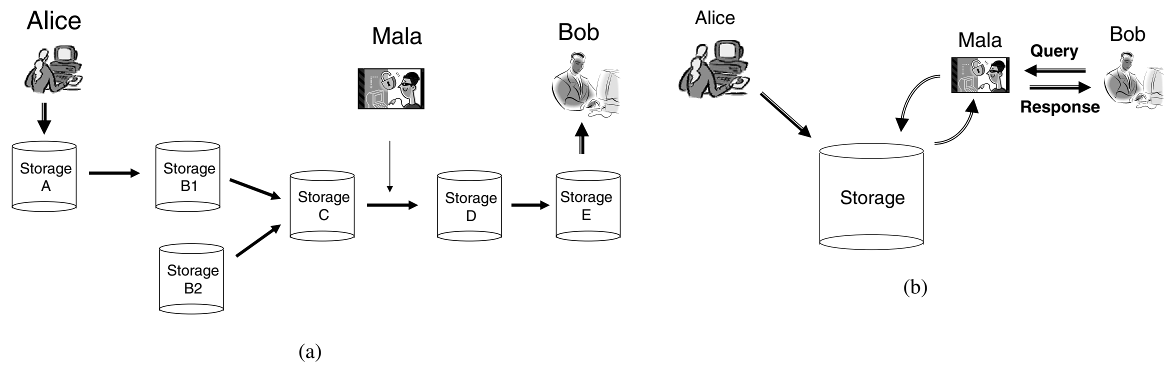


Figure 1. (a) Typical Server-to-Server Migration Scenario. Alice commits a file on server A. The file is then migrated through $B1$, C and D. Finally, Bob queries the file from E. Adversary Mala could tamper with any of these migrations. (b) Query Result Migration. Remote user Bob submits a query to Mala, who obtains the query results and "migrates" them to Bob.

from the compliance server to a user in response to a query (query result migration), as shown in Figure 1. A **server-to-server** migration involves copying the contents of the source storage server S_A to S_B. However, the contents of S_B may not be an exact copy of S_A, as files may be renamed or moved from their original location. For example, the contents of the old server might be placed under the /archive/S_A directory of S_B. Further, there may be legitimate reasons to omit certain files during migration. For example, during a corporate spin-off, only documents relevant to the new company may be copied to the destination storage server. In such cases, we assume that a corporate *migration policy* determines the subset of files that must be copied to the destination device.

In **query result** migration (Figure 1(b)), Bob submits a query and receives a set of files in response. His query plays the same role as a migration policy. If Bob does not have a direct hardware connection to the compliance server, he faces threats identical to those of server-to-server migration: Mala can try to omit or alter files during migration from the storage server to Bob.

Query Model Suppose Bob is looking for a file or directory R. Bob must learn the server name and the path under which R was originally committed. Bob will typically obtain this information using a trustworthy index structure [20]. For example, if Bob wants all files containing the keyword "compliance", he may get the path S_A:/home/Alice/audit from a lookup in an inverted index. Or he may know a priori that he is only interested in files from S_A:/home/Alice/mbox/.

Security Guarantees We provide two types of security guarantees. The *integrity* guarantees enable a querier Bob to verify that a migration was carried out in accordance with a stated migration policy. The *secrecy* guarantees prevent Bob from obtaining information that he should not be able to access. For example, if the policy dictates the removal of a file R during migration, Bob should not be able to access R's contents or metadata on the new server. Our security guarantees are summarized below.

<u>Case 1:</u> *R has the same location (path) on S_B as on S_A.*

Integrity Guarantee: *Bob can verify that the contents of R's data or metadata have not been altered since they left S_A, other than data possibly having been appended to it on S_B.*

<u>Case 2:</u> *R's location has changed and the change to its path is recorded in the migration log.*

Integrity Guarantee: *Bob can determine R's new location on S_B. Also, as in the previous case, Bob can also verify the integrity of R's contents and metadata.*

<u>Case 3:</u> *R has been omitted during migration, or its location has changed and the change to its path is not recorded in the log.*

Integrity Guarantee: *Ordinary file system commands to open R or read its metadata will be met with the response "file not found", as for any deleted or nonexistent file. However, through a separate channel that we provide, Bob can validate that R's omission was consistent with the migration policy in effect at the time.*

Secrecy Guarantee: *Bob should not learn any metadata or data other than what may be required to validate the migration policy. For example, if the policy was to omit files expiring in the coming year, Bob can learn the expiry times of all omitted files but not their contents, size or any other metadata.*

Our guarantees do not cover every possible way that

Bob can learn about R's contents or metadata:

- **Index leaks:** When an index over the files is migrated to a new server, care must be taken to avoid the inclusion of information about omitted files in the index [22]. Such techniques are beyond the scope of this paper.

- **Partial metadata leak:** Under one of our proposed migration approaches, Bob can learn the metadata values that are required to validate the migration. Ideally he should only learn whether the metadata satisfied the migration policy, not the exact metadata values. For example, for the *expiry_date* based omission policy given above, Bob should only learn whether *expiry_date* was within the specified period, not the exact date. We also offer a migration approach that prevents such leaks, as discussed later.

- **Path leaks:** If the path to a file or directory is altered during migration (as described in Section 5.1), we require Mala to record both its old and new location in the migration log. The path to a file or directory may also be recorded in an index. Unless the index or the log is cleaned up, Bob can learn that a file with a given path existed, even after the file has been deleted. Securely cleaning up an index or the migration log is a non-trivial problem and is left as future work. In this paper we assume this path is not sensitive. The other option is to store the path encrypted, as briefly outlined below.

Every file and directory in the system is assigned a symmetric encryption key, used to encrypt the path. Every occurrence of the path (e.g., in the log) is hierarchically encrypted – the top directory is encrypted with the top directory specific key, the subdirectory under it with a subdirectory specific key, and so on. For example, if the complete path to a file is $/A/B/C$, it is encrypted as $E_{ka}(A)/E_{kb}(B)/E_{kc}(C)$, where ka, kb and kc are the keys associated with A, A/B and $A/B/C$. The encryption (decryption) key is stored with the file/directory itself as metadata and is deleted when the file/directory is deleted.

To decrypt the path, Bob reads in the corresponding decryption keys from the file metadata, starting from the top directory. After a directory is deleted (and hence the decryption keys of all the files/directories under it are deleted), Bob cannot decrypt the paths of the files under it. Additional measures like padding the encrypted paths to a default length can be used to prevent Bob from guessing the depth of the directory hierarchy under the deleted directory. Bob can still decrypt the portion of the path corresponding to the parent directories of the deleted directory. Bob hence can learn the number of files created under the deleted directory. We assume that this is not a serious threat. A detailed version of the this scheme is available in the longer version of the paper.

For query result migration, Bob must be able to verify that his query ran correctly on the server and the query results were not altered during shipment.

Notation Our notation is summarized in Figure 2. Suppose the path of a file or directory f is $/f_1/\ldots/f_n$. We will often write this path as f^p/f^r, where f^p is the path $/f_1/\ldots/f_{n-1}$ to the parent of f, and f^r is the remaining component f_n of the path. For example, for the file **/emails/alice/mbox**, f^p is **/emails/alice** and f^r is **mbox**. Additionally, we use the terminology *path*, *directory path* and *relative file name* to refer to f^p/f^r, f^p, and f^r, respectively. For brevity, we will often use the term *file* f to refer to the path to f.

3. Related Work

The problem of securing data on a storage server has received a lot of research attention. One of the earliest works in this direction introduced cryptographic file systems [5], in which all the files and directories are encrypted with a user-provided key. Subsequent research addressed untrusted storage servers [12, 13, 19, 4], or provided security even against superuser attacks [27]. These systems were designed to secure data on a single storage server and not across migration. The most important issue that differentiates compliance storage from single-server security is that we cannot trust the original owner of the data (who has the file secret key). An adversarial owner can tamper with a record during migration and re-encrypt it with the secret key. We also cannot rely on integrity checkers like Tripwire that periodically compare file hashes against a trusted source of known file hash values. In our setting, these known file hash values must themselves be maintained on the storage server and migrated along with the data. An adversary can alter these stored hash values during migration and make them consistent with the tampered files, thus avoiding detection by Tripwire.

A problem of binding files to their positions in the directory hierarchy has been studied before in SFS-RO [12]. SFS-RO however, is a read only file system and does support file creations, appends or namespace alterations. Randal et. al. proposed a system for providing verifiable audit trails for versioning file systems [7]. In their scheme, a set of published MACs lets a querier verify the authenticity of current and older versions of the filesystem. Unlike a versioning file system, the querier in our case has access to only the latest file/directory version on the current storage server. Based on this current version and a set of old file signatures, the querier must verify that the current version has been obtained through a series of proper migration operations.

$h(f)$	A cryptographically secure hash function acting on data (usually the contents of file) f.
$\langle x_1, \ldots, x_n \rangle$	Ordered sequence $x_1, \ldots, x_n$.
$h^t(\langle x_1, \ldots, x_n \rangle)$	A cryptographically hash function acting on a tuple $(x_1, \ldots, x_n)$.
K and K^{-1}	Public and private keys of a public key cryptosystem.
$\{\}_K$ and $\{\}_{K^{-1}}$	Encryption and decryption with public and private keys respectively.
$C_x[f]$	Certificates of type x that can be looked up by path f.

Figure 2. Notation

In this paper, we also explore the problem of trustworthy, secrecy preserving, policy driven file deletion. The problem of deletion of file system data from backup copies [6] and versioning file systems [25] has been explored before. Their goal, however, was to securely delete data from multiple copies (using cryptographic schemes), rather than to authenticate the deletion operation for subsequent queries. These works assume that the person invoking the deletion is trustworthy. We also explore the problem of secure namespace alterations which to the best of our knowledge hasn't been studied before.

The problem of securely migrating a query result is closely related to the problem of query execution for outsourced databases [9, 14, 24, 29]. In database outsourcing, the server itself is untrusted, the surrounding environment and users are trusted, the goal is to ensure correct answers to all possible supported queries, and the data publishers sign the data and indexes beforehand. We trust the server and expect it to create certificates on the fly in response to the migration query. This flexibility enables us to handle all possible migration queries, unlike the database outsourcing work where only a restrictive subset of SQL queries is supported.

One of our migration schemes requires running the query engine inside the storage server. The idea of downloading parts of a query engine into the storage server was proposed in research on active disks [26, 3], although the goal there was to improve query performance. In our case, the query can run slowly; our goal is to ensure that its answer is correct and verifiable.

4. Infrastructure for Migration

For trustworthy migration, we enhance the storage server so that it can sign any file or directory and return certificates attesting to its contents and metadata. These certificates let the querier Bob verify that the file/directory contents or metadata have not been tampered with during migration. These certificates are only generated at migration time, i.e., they are not generated or accessed during ordinary file system write operations.

Specifically, we add three functions, *SG_M*, *SG_C* and *SG_D*, to the access protocol (e.g., NFS) supported by the storage box. *SG_M* takes a file f (its complete path) and a list of metadata fields (meta-list) as argument and returns a tuple consisting of i) hash of its full path $h^t(f^p/f^r)$ ii) meta-list, iii) the hash h^t(list-metadata), where list-metadata is the ordered list of f's metadata values corresponding to the fields listed in meta-list and iv) the current time ts from the storage server S_A's reliable clock—signed with the server's private key $K^{-1}{}_{S_A}$. In this paper, meta-list is a subset of the metadata (*create time*, *owner*, *expiry time*, *commit server*). For example, if meta-list=$\langle$*expiry time, owner*$\rangle$, then *expiry time* and *owner* of f are included in list-metadata. As we discuss later, those metadata fields that must be preserved while migrating a file f are passed as the argument meta-list.

SG_C(f) returns the hash of f's path, the hash of its contents ($f.contents$) and the current time ts, signed with the server's private key. Finally, *SG_D* returns the signed hash of the directory contents—the sorted list of the relative paths (f^r) or subdirectories within the directory. We use the notation C_{meta}, C_{data}, and C_{dir} to refer to the certificates returned by the functions *SG_M*, *SG_C* and *SG_D*, respectively. We use the notation $\{C_x\}_{K_{S_A}}[i]$ to refer to the ith field of a certificate. For example, the 0th field is the hash of the path.

Our threat model says that Mala does not regret the existence of a particular document before it has been committed to the server. This does not rule out the possibility that Mala believes that some day there may be some document, currently unspecified, that she would like to hide. We use a timestamp in certificate generation so that Mala cannot systematically pre-generate certificates for files and directories long before they are migrated. When Bob wants to verify a file on S_B, he will look up the public key and migration start time for S_A from a trustworthy source.

5. Trustworthy Migration

We first consider the case where the entire directory structure is copied intact from S_A to S_B. To migrate a file with path f, the migrator Mala must invoke *SG_M*$(f$, all-metadata) and *SG_C*(f) on S_A and store the returned certificates C_{meta} and C_{data} on S_B. Here all-metadata is $\langle$*create time*, *owner*, *expiry time*, *commit server*$\rangle$. Similarly, for a directory d, she must generate C_{meta} and C_{dir} and copy them to S_B. The certificates must be stored in a

Function	Runs on	Implementation
$SG\_M(f,\text{meta\_list})$	File/Directory	$\{\langle h^t(f^p/f^r),\text{meta\_list},h^t(\text{list-metadata}),ts\rangle\}_{K_{S_A}^{-1}}$
$SG\_C(f)$	File	$\{\langle h^t(f^p/f^r),h(f.contents),ts\rangle\}_{K_{S_A}^{-1}}$
$SG\_D(d)$	Directory	$\{\langle h^t(d^p/d^r),h(\text{d}.contents),ts)\rangle\}_{K_{S_A}^{-1}}$

Figure 3. Storage Functions

```
VerifyFile(f)    // Checks f against certifi cate

 1: Let $f^p/f^r$ be $f$'s path
 2: $phash = h^t(f^p/f^r)$
 3: if !(($phash == \{C_{meta}[f]\}_{K_{S_A}}[0] == \{C_{data}[f]\}_{K_{S_A}}[0]$) then
 4:     return INVALID_MIGRATION
        {The certifi cate does not correspond to this fi le }
 5: end if
 6: $M$=$f$.metadata {$f$'s metadata on destination $S_B$}
 7: $(own,crt,exp,csrv)$=
    ($M$.owner,$M$.create_date,$M$.expiry,$M$.commit_server)
 8: $mtuple = \langle own,crt,exp,csrv\rangle$
 9: if ($h^t(mtuple) \neq \{C_{meta}[f]\}_{K_{S_A}}[2]$) then
10:     return INVALID_MIGRATION
        {$f$'s metadata on $S_B$ does not match with certifi cate}
11: end if
12: $sz = M$.migration_size {The size of the fi le when it was migrated. Stored
    as metadata.}
13: $cont\_hash = h(f$.contents[0..$sz$]) {Hash the fi rst $sz$ bytes of $f$}
14: if ($cont\_hash! = \{C_{data}[f]\}_{K_{S_A}}[1]$) then
15:     INVALID_MIGRATION;
16: end if
17: return VALID
```

```
VerifyDir(d)    // Checks d against certifi cate

 1: Let $d^p/d^r$ be $d$'s path
 2: $phash = h^t(d^p/d^r)$
 3: if !(($phash == \{C_{meta}[d]\}_{K_{S_A}}[0] == \{C_{dir}[d]\}_{K_{S_A}}[0]$) then
 4:     return INVALID_MIGRATION
 5: end if
 6: $M$=$d$.metadata
 7: $(own,crt,exp,csrv)$=
    ($M$.owner,$M$.create_date,$M$.expiry,$M$.commit_server)
 8: $mtuple = \langle own,crt,exp,csrv\rangle$ {Create the tuple}
 9: if ($h^t(mtuple) \neq \{C_{meta}[d]\}_{K_{S_A}}[2]$) then
10:     return INVALID_MIGRATION
11: end if
12: for all ($f \in d$.fi les) do
13:     if ($f$.commit_server == $S_A$) then
14:         $dir\_list$.append($f^r$)
15:     end if
16: end for
17: $hash = h(dir\_list \bigcup DELETE\_LIST(d))$
18: if ($hash \neq \{C_{dir}[d]\}_{K_{S_A}}[1]$) then
19:     return INVALID_MIGRATION;
20: end if
21: return VALID
```

Figure 4. Validation Routines

manner that allows them to be looked up by file/directory paths (f) on S_B. On S_B, we refer to these certificates as $C_x[f]$ where, $x \in \{data, meta, dir\}$.

While reading a file with path f on S_B, Bob can validate that its metadata and content have been preserved across migration by hashing its metadata and content on S_B and validating it against the signed hashes stored in $C_{meta}[f]$ and $C_{data}[f]$ (or $C_{dir}[f]$ if f is a directory). The pseudocodes for this operation appear as VerifyFile() and VerifyDir(d) in Figure 4. $C_{dir}[f]$ also lets one verify the completeness of migration: if a particular file under directory d is omitted inappropriately, the hash computed on the names of files under d will not match the hash stored in $C_{dir}[d]$.

We allow data to be appended to existing files, and new files to be created in migrated directories on S_B. To validate a file or directory f against the certificate generated by S_A, Bob must compute the hash over the content that was migrated from S_A, excluding additional data committed on S_B. For this purpose, we record the name of the server where f was originally committed, and the file size in effect at the time of its migration. The file size f.migration_size is used to compute the hash over the portion of the file that was committed on S_A (lines 12-14 of VerifyFile). Similarly, for directories, only those files and sub-directories whose commit server metadata is S_A are included while calculating the hash over the directory contents (lines 13-15 of VerifyDir). The migration time version of any other metadata that can be updated on S_B (for example *last access time*) must also be recorded.

The above migration approach can be extended to handle file omissions too. Consider a metadata based file omission policy, for example, where files expiring within a specified date are not copied to the destination storage. Let f be the path of a file meeting this omission policy. Mala must execute the following steps for f during migration:

- Invoke $SG\_M(f, \langle$ expiry date $\rangle)$.
- Store the returned certificate $C_{meta}[f]$ on S_B. Bob should be able to look up the certificate based on f's path on S_A.

- Record f's path and its *expiry date* in the migration log file (as a DEL operation, introduced later). No other data or metadata of f is copied to S_B.

When Bob attempts to access file f on S_B, he gets a "File Not Found" response. To verify if f's omission was legitimate, he can verify that the expiry-date metadata stored in the log satisfies the omission policy given in the log. Furthermore, Bob can validate that the stored expiry date is f's correct expiry date and not Mala's fabrication, by comparing it to the signed expiry date stored in $C_{meta}[f]$.

5.1. Namespace Alteration

In this section, we generalize the previous migration scheme to handle namespace changes and file omissions during migration. First, we define a set of basic operations, SNAP, CREATE, COPY and DEL, such that any migration involving file omissions and rearrangements is logically equivalent to a composition of operations from this set. As a part of the migration activity, we require Mala to create and store a migration log file $\mathscr{L}$, consisting of the migration policy followed by the sequence of these basic operations that maps the directory hierarchy on S_A to S_B. We also require Mala to generate certificates attesting to all the files and directories on S_A, and store them in a manner that allows them to be looked up by their paths on S_B.

We then propose a framework that lets Bob verify that the directory tree on S_B can be obtained by applying the operations in the migration log to the directory tree on S_A. In other words, if Mala omits a file from S_A without recording it in the log, then Bob can detect this. If Bob is looking for a file f from S_A, he can verify whether f was omitted, renamed, or kept intact during migration, or whether it did not exist on S_A at all. Finally, we propose a framework to let Bob verify the consistency of the log operations with the corporate migration policy. For example, if a file is omitted during migration, Bob can determine whether the omission violated the policy.

5.1.1. Restructuring Operations Figure 6 shows an example of directory restructuring. A *migration log* $\mathscr{L}$ consists of one SNAP operation followed by a sequence of CREATE , COPY , and DEL operations on the directory tree, defined as follows.

- SNAP performs a complete copy of the entire directory tree from S_A to S_B, including all metadata.
- CREATE (d^r_{new}, d^p, $meta$) creates a new directory d^r_{new} under parent directory d^p on S_B, and associates the usual metadata items $meta$ with it. The parent directory d^p must exist on S_B when this operation is performed.
- COPY (f^r_{src}, d^p_{src}, f^r_{dest}, d^p_{dest}) copies the file or directory f^r_{src} from under the directory d^p_{src} on S_B to the directory d^p_{dest} with the new name f^r_{dest}. If d^p_{src}/f^r_{src} is a directory, the entire tree under it is copied. The metadata of the copied files and directories (other than the name if $f^r_{src} \neq f^r_{dest}$) is unchanged, and the original file/directory d^p_{src}/f^p_{src} is not deleted.
- DEL (f^r, d^p, $meta$) deletes the file or directory d^p/f^r, that is, the file with relative name f^r under directory d^p. The metadata $meta$ required to validate the deletion policy is also recorded in the log.

Given a particular source directory, many different sequences of restructuring operations can produce the same final directory structure. We claim that any such sequence is logically equivalent to a sequence in migration log format. The log format facilitates reasoning about the effect of the migration, and is not intended as a literal representation of migration activities. For example, Mala cannot execute DEL on WORM storage. Instead, she can simulate DEL by not copying the deleted file during SNAP . Mala can perform any sequence of restructuring operations that is logically equivalent to the sequence she records in the log. Any discrepancy between the two will be detectable by our verification routines. In this paper, we do not introduce a test for logical equivalence or examine the question of which migration logs are undesirable from some perspective (e.g., logs that create and delete the same file repeatedly, or copy the contents of one file onto another). For our purposes, it is sufficient that naming conflicts be resolved in one way or another and that Mala be capable of producing a log.

5.1.2. Log-Based Migration onto an Empty Server

During migration, Mala must create a migration log file that is logically equivalent to the migration operations she intends to perform, and store the log on S_B[1]. As in an exact-copy migration, Mala must also generate and store certificates for all the files and directories on S_A. Bob can use these certificates in conjunction with the migration log file to verify the integrity of a file or directory migrated from S_A.

Consider a directory d_A that was originally committed on S_A and migrated to d_B on S_B. To verify that d_A was migrated correctly, Bob will use the migration log to trace the movement of d_A and find it on S_B. The migration log can also be used to determine the set of files and directories created and deleted under d_A during the migration. This information can be used to reverse map the contents of d_B on S_B to those of d_A on S_A. By verifying the integrity of the

[1]If desired, the migration logs and certificates can be stored on any other compliance server accessible to users.

```
VerifyDirRec($d$, $add\_set$, $del\_set$, $\mathscr{L}$, $orig\_path$)

 1: while (($top = \mathscr{L}$.pop()) $\neq$ END) do
 2:    if ($top$ == COPY ($f^r_{src}$,$d^p_{src}$,$f^r_{dest}$,$d^p_{dest}$)) then
 3:       if ($d^p_{dest}$ == $d$ ) then
 4:          $add\_set$.Append($f^r_{dest}$)
 5:       end if
 6:       if (isprefix($d^p_{src}/f^r_{src}$, $d$)) then
 7:          $rel\_path$ =suffix($d^p_{src}/f^r_{src}$, $d$)
 8:          $new\_d = d^p_{dest}/f^r_{dest}/rel\_path$
 9:          $ret$=VerifyDirRec ($new\_d, add\_set, del\_set, \mathscr{L}, orig\_path$)
10:          if ($ret \neq$ DELETED) then
11:             return ret
12:          end if
13:       end if
14:    end if
15:    if ($top$ == DEL ($f^r$,$d^p$,$meta$)) then
16:       if (isprefix($d^p/f^r$,$d$)) then
17:          return DELETED
18:       end if
19:       if ($d^p$ == $d$) then
20:          if ($f^r \notin add\_set$) then
21:             $del\_set$.Append($f^r$)
22:          else
23:             $add\_set$.Erase($f^r$)
24:          end if
25:       end if
26:    end if
27:    if ($top$ == CREATE ($d^r_{new}, d^p, meta$)) then
28:       if ($d^p$==$d$) then
29:          $add\_set$.Append($d^r_{new}$)
30:       end if
31:    end if
32: end while
33: if ($d$.not_exists) then
34:    return DELETED
35: end if
36: $dir\_list = \langle\rangle$; {Now compute the hash properly}
37: for all ($f \in d$.files) do
38:    if ($f$.commit_server == $S_A$) then
39:       if ($f^r \notin add\_set$) then
40:          $dir\_list$.Append($f^r$) {Include in hash only if $f$ was not
          present}
41:       end if
42:    end if
43: end for
44: $phash = h^t(orig\_path)$
45: if !($phash == \{C_{dir}[d]\}_{K_{S_A}}$ [0]) then
46:    return INVALID_MIGRATION {Certificate does not correspond
   to the correct directory}
47: end if
48: $dir\_list$.Append($del\_set$)
49: $hash = h(dir\_list)$
50: if ($hash \neq \{C_{dir}[d]\}_{K_{S_A}}$ .[1]) then
51:    return INVALID_MIGRATION
52: end if{Also verify the metadata and path hash}
53: return VALID;

VerifyDir($d$)

 1: Let $d = d^p/d^r$
 2: return VerifyDirRec($d$,{},{},$\mathscr{L}$,$d^r$)
```

Figure 5. Log Verification Routines

reverse-mapped contents of d_A with respect to the certificate for d_A obtained from S_A, Bob can verify the integrity of the migration.

Figure 5 presents the routine VerifyDir for verifying the contents of a directory d. VerifyDir invokes VerifyDirRec, passing two variables add_set and del_set. These keep track of the set of directories and files that have been added to or deleted from d, as explained below.

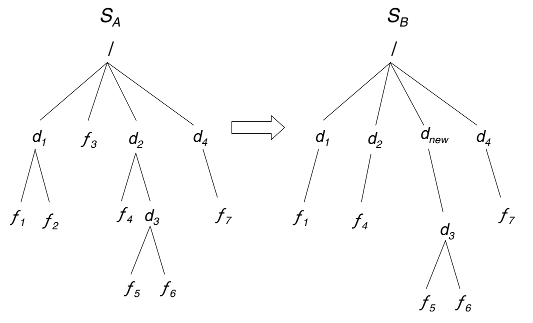


Figure 6. The sequence SNAP, DEL ($GID(d_1/f_2)$, $/d_1$, $meta(f_2)$), DEL ($GID(/f_3)$, '/', $meta(f_3)$), CREATE (d_{new}, /), COPY (d_3, $/d_2$, d_3, $/d_{dest}$) and DEL ($GID(/d_2/d_3)$, $/d_2$,$meta(d_2)$) is applied to the left directory tree to obtain the right one.

- If a file or directory is created under d (using COPY and CREATE operations), it is included in add_set. The entries of add_set are excluded when computing the hash of the contents of d_B on S_B (line 39 of VerifyDirRec).
- If a file or directory is deleted under d using the DEL operation, the deleted file/directory is added to del_set. The entries of del_set are added to the contents of d_B on S_B, to produce its contents on S_A (line 48).
- If d is copied to a new location new_d, its contents can be recovered from both its current and its new locations. Hence VerifyDirRec() is invoked on the new location as well (line 9). The original path to d is also passed as an argument to VerifyDirRec(), for validating the certificate.

The routine isprefix(d, f) (not defined in Figure 5) returns true if d is an ancestor of f (or is equal to f) in the directory hierarchy. The routine suffix(d, f) returns the remainder of the path to f once one has reached d. For example, isprefix(/A/B/, /A/B/C/D) is true, while suffix(/A/B/, /A/B/C/D) is C/D.

The security of the scheme can be argued as follows. If Mala omits a file f in directory d during migration, but does not record that in the migration log, then Bob's reconstruction on S_B of d's contents on S_A will not include f, and the validation of d against the certificate generated on S_A will fail. The same argument holds after d is copied into a new location during migration, since we validate d's contents with the original certificate for d. If Mala creates an illegal log—e.g., she copies d into a nonexistent directory—Bob's validation check will detect that where the directory contents are reconstructed. Any undefined operation in the log indicates an invalid migration and will lead to automatic presumption of guilt.

The pseudocode for verifying the integrity of a file is simpler. The log is used to trace the movement of the file across migration. The file contents are validated with the certificate for f obtained from S_A.

5.1.3. Other Metadata Changes We can generalize the migration log introduced in the previous section to handle arbitrary metadata changes during migration. Metadata changes applied to f can be recorded in the migration log by introducing a new log operation $\mathsf{MetaChange}(f, old\_meta, new\_meta)$, which records f's old and new metadata. These log entries can be used to reverse map the metadata of a file on S_B to that on S_A, while validating the certificate.

5.2. Special Cases

Migration onto a Nonempty Server Naming conflicts can arise when the contents of S_A are migrated onto a nonempty server S_B. If a file or directory f on S_A has the same path as a file already present on S_B then Mala must place f in a different location during migration. Figure 7(a) gives such an example, where the file $/f_1$ on S_A conflicts with $/f_1$ already on S_B, and hence is given a new name f_{new}. While verifying the contents of the directory on S_B against the certificate signed by S_A, Bob can use the *commit server* metadata to identify the portion of the directory tree that was migrated from S_A. For example, Bob uses this metadata to determine that only f_1 and d_1 were migrated from S_A.

Multiple Migrations We now consider the case where a file or directory is migrated several times before Bob accesses it. Figure 7(b) shows directory d as it is migrated from S_A to S_B and from S_B to S_C.

In a multiple migration scenario, all the certificates of a file must also be migrated. Each migration stores its own log plus the logs from all previous migrations, and also generates and stores a certificate for d's content. Thus two certificates are stored with d on S_C in Figure 7, signed by S_A and S_B, respectively. If Bob wants to read d, he should validate d against both certificates. When validating against S_A's certificate, Bob must consider only those files/directories which were committed on S_A; when validating against S_B's certificate, both S_A's and S_B's content must be included. Furthermore, paths of files that are created on S_A and are subsequently deleted on S_B (for example on expiry) must be recorded in the log, as a DEL operation. Those must also be included while reconstructing d's contents on S_A based on its contents from S_C.

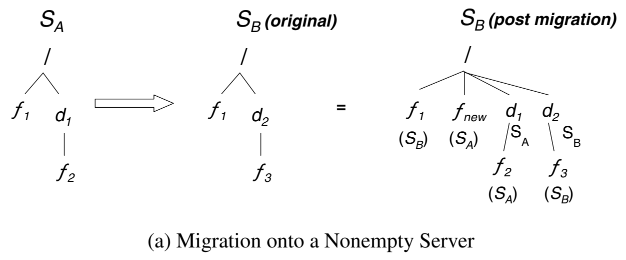
An ordinary file f may have had data appended to it on each storage server. The migration time sizes of f, $size_A$ and $size_B$, are both kept as metadata for f. Bob can validate f's contents by computing the hashes over the first $size_A$ and $size_B$ bytes of f, and comparing them against the certificates from S_A and S_B, respectively.

The above scheme can be extended to the case where the directory structure is also modified during each migration step, by considering the two logs $\mathscr{L}_{AB}$ and $\mathscr{L}_{BC}$ in conjunction. Suppose Bob queries for the directory d that was committed on S_A and wants to verify that its contents from S_A have been properly migrated. Bob can trace the set of operations applied to d, using the concatenation of the logs. Furthermore, while reconstructing d's contents on S_A using d's contents on S_C (to validate it against the certificate signed by S_A), Bob must consider only the files or directories under d with S_A listed as their original storage server. To verify a directory committed on S_B, Bob must use the migration log $\mathscr{L}_{BC}$ and consider files or directories under d with original server S_A or S_B.

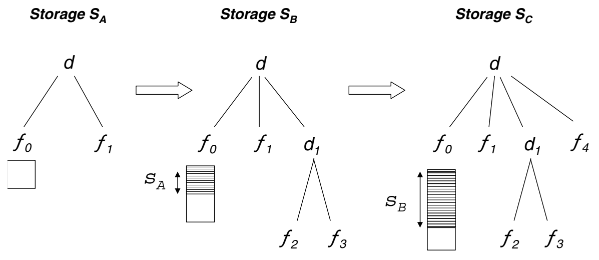
Retaining all the certificates for a file raises the threat of information leakage from older certificates. Consider a file f that was created on S_A, migrated to S_B and omitted during migration to S_C, say based on its expiry date. In order for Bob to verify that f's expiry date has not been tampered with during migration, f's certificates signed by S_A and S_B must be stored on S_C. However, unlike the S_B certificate, the S_A certificate was obtained when f was migrated to S_B and hence includes all of f's metadata. We can handle this by precomputing an expiration certificate for a file each time that it is migrated. The expiration certificate includes only the file's initial commit path and expiry date, which is all that is needed to validate its eventual omission and thus is the only certificate that needs to be stored for f on S_C. Precomputing the certificate is feasible only when the omission policy is known in advance. The other option of precomputing separate certificates for each of the metadata fields is highly space inefficient.

An alternate solution is to encrypt the metadata fields before including them in the certificate (this is similar to our path encryption idea). Specifically, $C_{meta}[f]$ can include the hash of the encrypted metadata fields — $h^t\langle E_{ko}(\text{owner}), E_{kc}(\text{create\_date}), E_{ke}(\text{expiry})\rangle$ instead of hash of the plain text metadata. ko, kc and ke are symmetric keys used to encrypt the owner, create_data and expiry date metadata fields respectively. The encryption keys are stored with the file/directory as a metadata.

File omissions are handled by migrating the encrypted metadata fields of the omitted file to the destination storage. In addition, the decryption keys of the metadata fields required to validate the omission (e.g. expiry_date in the



(a) Migration onto a Nonempty Server



(b) Multiple Migrations

Figure 7. (a) The contents of S_A are migrated onto S_B (original), which already contains files with similar names. File f_1 on S_A is renamed to f_{new}. The original storage server is noted in the metadata for each file. (b) Directory d is migrated from S_A through S_B to S_C. The metadata associated with file f_0 on S_C includes the server S_A where it was originally committed, and its sizes $size_A$ and $size_B$ at the time of each migration. On S_C, the originating server S_B is recorded in the metadata for d_1, f_2, and f_3.

example above) are copied. The encrypted fields lets a querier verify the last certificate (generated by S_B in the example above) against the old certificates (S_A's certificate). Using the migrated keys, the querier can decrypt the metadata fields (and only those fields) that are required to validate the migration.

The above scheme is space inefficient since it requires a separate key to be maintained for each metadata field. We can address this by generating the metadata specific key from a single shared key and the unencrypted contents of the metadata. For example, the key *ke* used to encrypt the expiry_date metadata can be generated as $k \oplus h(expiry\_date)$. We discuss this scheme in details including its security properties in the full version of the paper.

6. Migration Policies

Metadata-based migration policies can be validated using the log. Since we store the relevant metadata of each omitted file in the migration log file as an argument to the DEL operation, Bob can verify whether a file or directory satisfies this omission policy. Bob can validate the metadata stored with the DEL operation by comparing it to the signed metadata hash in $C_{meta}[f]$.

Complex migration policies may be needed, for example, during a company spinoff. Only the "relevant" data from the original company's storage server is to be migrated to the new company's server, and relevance depends on the file contents. For example, a policy may state that all email documents where either the sender or receiver is an employee of the new company must be migrated. The sender/receiver information is available only by scanning document contents.

We treat a complex migration policy as a query executed over the storage server, such that the result of the query is the set of files that must be migrated. Bob can tell whether the migration policy was satisfied for a particular file or directory d by examining the query result and determining whether the file is present on the target server. This raises a key issue: how can Bob verify the integrity of the query result? In other words, why should Bob believe that the query result was obtained by running the query on the set of files on the original server, and is not just Mala's fabrication? If we solve this problem, we also have a solution to the problem of delivering a query result to a remote user. We propose two solutions to address this problem.

Input-centric Policy-based Migration The key idea of input-centric migration is to migrate the read set of the query along with the query result. We define the *read set* to be all the data read by the query engine while processing the query defining the migration policy. For example, suppose that the query is *Migrate all documents containing the keyword "X"*, and an inverted index is available to answer the query. Generally, we expect each posting list in the inverted index to be stored in a separate file. Thus the read set includes the posting list for the term "X", along with the files containing any higher-level index structures used by the query engine to access X's posting list. These may include the dictionary, the map between dictionary words and pointers to the posting list, and the map from document IDs to document locations. If Mala had used a B-tree to execute her query, then the read set would have contained all the nodes traversed in the tree (assuming separate nodes are stored in separate files).

To perform a migration, in addition to her usual tasks, Mala runs the query Q using the query engine QE and migrates the read set and the files corresponding to the query result to the destination server. Additionally, she must invoke *SG_M* and *SG_D* on all the files and directories constituting the read set and make those certificates accessible to Bob. Finally, she records the fact that the query engine QE was used with query Q to migrate from S_A to S_B, together with the timestamp for the start of migration.

To verify the migration, Bob obtains the query engine QE and the query Q. To gain trust, Bob can inspect the query engine code to determine that it will execute the query properly. Alternatively, Bob can run a trusted version of the query engine certified by an appropriate organization. Bob must verify that the read set was migrated correctly based on the certificates signed by S_A, and determine that the read set has been set up properly for the query engine to find it. Finally, Bob can run the query engine QE with query Q on the migrated read set to regenerate the query result. He can use the query result to verify that the files omitted during migration satisfied the corporate omission policy.

The input-centric approach is infeasible if the read set is extremely large (e.g., all documents on the server must be scanned for a keyword) or the query engine will not be portable for the lifetime of the migrated documents. The read set can also leak information. For example, Bob might be able to reconstruct the contents of a deleted file from the read set [22]. To address these problems, we propose an output-centric approach to migration in the next section.

Output-centric Policy-based Migration The key idea behind the output-centric approach is to run the query engine inside the trusted environment of the storage server, and have it sign a certificate of execution that testifies to the code used, the arguments passed to it, and the query result produced. The certificate, together with a small amount of auxiliary information, allows Bob to verify the correctness of the migration at any future point.

More precisely, we extend the storage server interface so that it can accept a program V from an external user (subject to the usual authorization controls), along with arguments $x_1,\dots,x_n$. The server executes V with arguments $x_1,\dots,x_n$, and returns a certificate of the form $\{h^t(V.code, x_1,\dots,x_n,r)\}_{K^{-1}}$, where h^t is a one-way hash function (e.g., SHA1 or its successors) acting on the tuple $(V.code, x_1,\dots,x_n,r)$; $V.code$ is the source code of program V; r is the value returned by executing $V(x_1,\dots,x_n)$; and K^{-1} is a private key known only to the storage server. The certificate is a proof that V was run with arguments $x_1,\dots,x_n$ on the server and the result r was produced.

Consider the following migration policy: *Migrate only those files containing terms in* keyword_set. The query implementing this policy is given by PolicySatisfy in Figure 8. It takes a file path as argument and returns true or false depending on whether the file has a word from keyword_set.

To start the migration, Mala downloads and runs PolicySatisfy on every file f on the system. After running PolicySatisfy(f) the server returns the following certificate:

```
PolicySatisfy($f$)    // Does $f$ satisfy the policy?

1: fkeys=$f$.Parse() {We assume that keyword_set is hard-coded in the
   function}
2: if (Intersect(fkeys,keyword_set)) then
3:     return true
4: else
5:     return false
6: end if

Validate($C$, $f$, PolicySatisfy, $K_A$)

1: Let $f^p/f^r$ be $f$'s full path.
2: $cert = h^t$(PolicySatisfy,$f^p/f^r$,false)
3: if ($cert == \{C\}_{K_A}$) then
4:     return SUCCESS
5: else
6:     return FAIL
7: end if
```

Figure 8. Query Code

$$C[f] \quad = \quad \{h^t(\text{PolicySatisfy}, f, \text{true/false}\}_{K_A^{-1}}$$

For files which satisfy the migration policy (i.e. have a word from keyword_set) and have to be migrated, Mala invokes *SG_M*, *SG_C* and *SG_D* as before and stores the returned certificate on S_B. For every other file f, Mala copies the above execution certificate $C[f]$ to S_B. As before, she creates a migration log, recording all the name space alterations and the migration policy. She also records PolicySatisfy and h^t in the log.

Bob can verify that a file omitted during migration indeed did not have any word from keyword_set by checking the certificate C. For this, he must obtain the appropriate version of PolicySatisfy and h^t from the log and the public keys of the secure coprocessor from a trusted key escrow service. To gain trust, Bob inspects the code of PolicySatisfy to determine that it implements the omission policy appropriately. He validates all this against the certificate C, by running the Validate code in Figure 8. This scheme is secure because if Mala logs a different version of PolicySatisfy than the one she actually ran, then C will not match the hash computed in line 3 of Validate. If Mala illegitimately omits a file during migration, she will not be able to produce a correct execution certificate for that file. In this approach, Bob can trust that an omitted file did not have a keyword from keyword_set, without learning the contents of the omitted file. The same idea, when implemented for metadata based migration can prevent metadata leaks.

Scheme	Type	Description
Metadata based	Migrated Files	$C_{meta}[f]$, $C_{data}[f]$, commit server, migration time sizes
	Omitted Files	DEL log entry, $C_{meta}[f]$ computed over enough metadata to validate omission
	Other	COPY and CREATE log entries, migration policy
Input Centric	Migrated Files	$C_{meta}[f]$, $C_{data}[f]$, commit server, migration time sizes
	Omitted Files	DEL log entry
	Other	read set, COPY and CREATE log entries, migration policy, query code
Output Centric	Migrated Files	$C_{meta}[f]$, $C_{data}[f]$, commit server, migration time sizes
	Omitted Files	DEL log entries, Execution Certificate $C[f]$
	Other	query code, COPY and CREATE log entries, migration policy

Figure 9. Objects that must be maintained for each file under the different migration schemes

7. Performance Evaluation

Trustworthy migration imposes no runtime performance penalty on file write operations. It does impose a space overhead for storing certificates and logs at destination servers, a time overhead for generating certificates during migration, and migration validation costs incurred by an auditor (or a reader who does not trust the auditor) certifying the results of a migration. Readers who trust the auditor incur a one-time cost for checking the auditor's migration certificate for a new server. Some schemes require generation of a certificate when a file is deleted. We measure these overheads below.

Space Overhead for Migrated Files The extra storage required for each migrated and omitted file is summarized in Figure 9. We calculate the size of $C_{meta}[f]$ as follows. (1) 128 bits collision resistant hash of file/directory path. (2) 32 bits for a *meta-list* bitmap, assuming up to 32 metadata items per file. (3) 128 bits for the metadata hash. (4) 64 bits for the timestamp ts. (5) 1024 bits for the signed hash, assuming a 1024 bit RSA. (6) 16 bytes for additional metadata such as the ID of the commit server. Thus an upper bound on the size of $C_{meta}[f]$ is 188 bytes. Similarly, we computed the size of $C_{data}[f]$ as 168 bytes. This space overhead can be reduced by storing only one copy of the path hash and timestamp ts instead of two in $C_{meta}[f]$ and $C_{data}[f]$. Overall, the certificate storage space per file is under 340 bytes per migration. We also record the *migration time size* (8 bytes per migration) and *commit server* (4 bytes) of each migrated file. The total space overhead comes to less than 5%, given today's average file size of 10-100 KB [10, 18]. As average file sizes are likely to increase in the future, due to the proliferation of multimedia data, our approach hence incurs a very reasonable space overhead.

Files omitted during migration require a tombstone on the destination storage. In the case of files that were migrated, subsequently expired, and then were deleted, this tombstone should be computed by the time the file is deleted. For the metadata-based migration scheme, this tombstone includes $C_{meta}[f]$ (188 bytes), selected metadata fields for policy validation, and a DEL log entry containing the file path (32 bytes). For the input approaches, only the DEL log entry is required. For output centric migration, the tombstone constitutes the query execution certificate. In either case, the total size for this tombstone is likely to be much smaller than the file itself. Thus, if the migration involves a small number of file omissions, the tombstone space overhead will be negligible compared to the total migrated data. Migrations with large numbers of file omissions too are often structured, for example, where a directory and all the files under it are deleted based on the directory's metadata. Such cases also require a small number of per-directory tombstones.

In addition to the per file items, our migration schemes also require three additional items to be stored. First, each CREATE and COPY log entry must be recorded. Most real life migrations are likely to involve only a few of these directory restructuring operations, such as moving "/" to a subdirectory. Second, the read set and query code must be recorded for input-centric migration. Typically the read set will be a subset of an index structure used to answer the query. If the read set is large, output-centric migration should be used instead. Finally, the log must record the omission policy, and the query code for output-centric migration. The overhead for these two log entries is independent of the size of the migrated files and should be modest in practice.

Time Overhead for Migration Trustworthy migration imposes costs for generating certificates at migration time. To evaluate this cost, we implemented the signing functions *SG_C* and *SG_M* using open source hashing and encryption libraries. As test data, we used the Enron email corpus [1]. It contains approximately 500,000 email messages from 150 users, mostly senior management of Enron, with each message stored in a separate file. We carried out the experiments on a single processor Pentium Xeon 1.3

Function	Enron Corpus (in sec)	Per File (in sec)
1. Path Hash	12	2.3×10^{-5}
2. File Hash	9145	0.017
3. Metadata Hash	72	1.39×10^{-4}
4. RSA Sign	3300	0.006
SG_M (1+3+4)	3385	0.006
SG_C (1+2+4)	12458	0.024

Figure 10. Run time for the fi le data and metadata signing operations

GHz machine with 512 MB of main memory. The emails were stored in an ext2 file system on an 80 GB, 5400 RPM, Maxtor 98196H8 drive with a 9 msec average seek time. Figure 10 reports the time overheads for trustworthy migration: computing path hashes, obtaining the hash over content/metadata, and signing the hash. As expected, hashing the contents and the public key signing constitute the major component of the total run time. The signing overhead per migrated and omitted file is 30 msec (cost of *SG_M* and *SG_C*) and 6 msec (*SG_M*) respectively.

All three migration approaches incur costs to determine which files to omit; these costs are the same as for non-trustworthy migration. Input-centric migration requires migration of all data touched during the process of determining which files to omit, which can be expensive if few files are to be migrated but much data was touched. Output-centric migration avoids this cost. Output-centric requires generation of migration certificates for each file.

Because migration is rare, the migration-time overhead to generate certificates will be a small component of the server lifetime workload. Furthermore, in practice, the old server is unlikely to be de-commissioned immediately. The certificates can be obtained by a process running on the old server after the new server is on line.

Time Overhead for Verification When the directory structure is migrated intact to a new server, the entire migration can be verified in the amount of time required to regenerate the file and directory certificates. A single file can be validated in 24 msec. These time estimates are also appropriate if the migration involves a modest number of directory restructurings, name space changes, and file omissions.

If a migration involves many file omissions, the log will be large. We can speed up directory verification by indexing each DEL operation in the log by the directory under which the operation is invoked. Then the verifier can find each omitted file for a particular directory in $O(1)$ time, as it only needs to consider DEL operations executed directly under d (not its descendants). The overall effect of omissions on validation time depends on what is omitted. If ordinary files are omitted but no directories, then validation of a metadata-based or input-centric migration will take approximately the same amount of time as the original process of selecting the files to omit and generating all the migration certificates. Validation of an output-centric migration will take approximately the same amount of time as generating the certificates for the migrated files.

The directory verification routine VerifyDir(d) calls itself recursively for every COPY operation involving d or any of its parents, halting the process when one of the calls succeeds. The worst case behavior occurs when n COPY operations are performed for n directories that are all ancestors and descendants of one another, and all copies are deleted later on in the log; in this case the overhead is $\Omega(2^n)$. However, such extensive and ultimately pointless directory restructuring is unlikely in a realistic migration scenario.

8. Conclusion

In this paper, we have proposed techniques for securely migrating data from one compliance storage device to another. Migration is carried out by generating a series of certificates attesting to the contents of the files and directory structure and creating a log of name space alterations. The certificates and the log allow any future user to verify that the migration steps satisfied the corporate migration policy. Together, these new facilities provide a trustworthy basis for decades-long retention of files containing compliance records.

Trustworthy migration imposes no performance penalty on file writes, which are the primary component of compliance workloads. Readers who trust an auditor incur a small one-time cost when using a new compliance server. Readers (auditors) who wish to verify migrated files on their own can do so at a rate of approximately 24 msec per file, based on our experiments with an Enron email data set. The additional cost at migration time is also approximately 24 msec per migrated file in our experiments. This is very reasonable, given that a typical compliance storage server will only be migrated once in its lifetime. Our migration approaches impose a space overhead of about 400 bytes per migrated file and, under some approaches, a 200-300 byte tombstone for each file that is not migrated. This is less than 5% overhead, given today's average file size of 10-100 KB.

Acknowledgments. This research was supported by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695 and by Sandia National Laboratories under grant number DOE SNL 541065.

References

[1] Enron email dataset. http://www.cs.cmu.edu/enron/.

[2] Recent fines of 1-25 million. www.facetime.com/pdf/reymann.pdf.

[3] A. Acharya, M. Uysal, and J. H. Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS*, 1998.

[4] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system, 2001. Unpublished report, available at citeseer.ist.psu.edu/batten01pstore.html.

[5] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1993.

[6] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security Conference*, 1996.

[7] R. Burns, Z. Peterson, G. Ateniese, and S. Bono. Verifiable audit trails for a versioning file system. In *Storage Security and Survivability Workshop*, 2005.

[8] Congress of the United States of America. Sarbanes-Oxley Act, 2002. Available at http://thomas.loc.gov.

[9] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic data publication over the internet. In *Journal of Computer Security*, volume 11, pages 291–314, 2003.

[10] A. B. Downey. The structural cause of file size distributions. In *SIGMETRICS/Performance*, pages 328–329, 2001.

[11] EMC. Centera Content Addressed Storage System. Available at http://www.emc.com/products/ systems/centera_ce.jsp.

[12] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *Computer Systems*, 20(1), 2002.

[13] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, 2003.

[14] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Second International Conference on Applied Cryptography and Network Security*, 2004.

[15] HP. StorageWorks Reference Information, 2006. Available at http://h18000.www1.hp.com/.

[16] L. Huang, W. Hsu, and F. Zheng. Content immutable storage for trustworthy record keeping. In *NASA MSST*, 2006.

[17] IBM Corp. IBM TotalStorage DR550, 2004. http://www-1.ibm.com/servers/storage/disk/dr.

[18] G. Irlam. Unix file size survey. 1993. http://www.base.com/gordoni/ufs93.html.

[19] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus — scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[20] S. Mitra, W. Hsu, and M. Winslett. Trustworthy Keyword Search for Regulatory Compliance. In *VLDB 2006*, Sept 2006.

[21] S. Mitra and M. Winslett. Trustworthy migration and retrieval of regulatory compliant records. Available at lily.cs.uiuc.edu/mitra1/migration.pdf.

[22] S. Mitra and M. Winslett. Secure Deletion from Inverted Indexes on Compliance Storage. In *Storage Security and Survivability Workshop*, Oct 2006.

[23] Network Appliance, Inc. SnapLock^{TM} Compliance and SnapLock Enterprise Software, 2003. Available at http://www.netapp.com/products/filer/snaplock.html.

[24] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, 2005.

[25] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin. Secure deletion for a versioning file system. In *FAST*, 2005.

[26] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB*, 1998.

[27] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM TISSEC*, 2(2), 1999.

[28] Securities and Exchange Commission. Guidance to Broker-Dealers on the Use of Electronic Storage Media under the National Commerce Act of 2000 with Respect to Rule 17a-4(f), 2001. Available at http://www.sec.gov/ rules/interp/34-44238.htm.

[29] R. Sion and B. Carbunar. Conjunctive keyword search on encrypted data with completeness and computational privacy. In *Cryptology ePrint Archive Report*, 2005.