Paranoia vs. Performance - A Quantitative Evaluation of Storage System Security

Erik Riedel, Mahesh Kallahalla, Ram Swaminathan

Storage Systems Program,
Computer Systems and Technology Laboratory,
Hewlett-Packard Laboratories, Palo Alto, CA

HPL–SSP–2001–6, Feburary 2001

Authorized for external release, October 2001

As users' and companies' dependence on shared, networked data continues to increase, the protection of this data from prying eyes or malicious hands becomes more and more critical. There are a variety of ways to ensure the security of data and the integrity of data transfer, depending on the set of anticipated attacks, the level of paranoia on the part of the data owners, and the level of inconvenience data users are willing to tolerate. This paper reviews all the previously proposed systems for secure, networked storage, and combines these ideas to describe a system for the "most paranoid" data owners. We show that the two main classes of systems in the existing design space are not really as far apart as they seem; one is essentially an optimization of the other. We outline the space of possible design choices and quantify the costs associated with each one using a trace from a time-sharing UNIX server used by a medium-sized workgroup. As we go along, we introduce a number of optimizations that have not been explicitly considered before.

Copyright © 2001 Hewlett-Packard Company. All rights reserved.

1 Introduction

Much of the focus of recent storage security work has been on protecting communication between clients and servers in a hostile, networked world [Gobioff98, Kent98, Mazieres99, Satran01]. The focus is on preventing snooping of data on the network, modification of requests in transit, and replaying of requests at future points. The most comprehensive treatment of this topic is the Network-Attached Secure Disk work [Gobioff99a], which protects the *integrity* (preventing unauthorized modification of commands or data) and, optionally, *privacy* (preventing the leaking of data in transit) of data transfers between clients and servers. The system uses capabilities provided to users by a file manager separate from the storage server which piggybacks the distribution of capabilities on namespace traversal. A major barrier to wide acceptance of the NASD scheme is the performance cost of the encryption and checksum operations required at both clients and servers. In order to reduce the cost of protecting integrity, the NASD designers proposed a scheme using partially-precomputed, hierarchical checksums with secure hashes [Gobioff99], but there is no comparable scheme to optimize privacy since this would require pre-computed encryption.

However, if data were stored on the server in already-encrypted form, then it would not be necessary to encrypt them for each transfer on the network. The difficulty with using such a scheme in NASD is that encryption is done using session keys generated for each client/server interaction, while pre-computation would require choosing a longer-lived key. From the client point of view, these two schemes are identical - it receives encrypted data and must pay the cost of decrypting it. From the point of view of an adversary, they are also equivalent - the data he sees is encrypted and unintelligible. The difference is only whether the server has to bear the encryption cost each time a new session key is chosen, or whether it can take advantage of data already stored in encrypted form and not have to doubly encrypt it.

As it happens, the storing of data in encrypted form on disk was originally proposed in Blaze's Cryptographic File System (CFS) and expanded in later systems [Blaze93, Cattaneo97, Zadok98, Hughes99], where it is used for a very different purpose - in order to protect data from untrusted servers. If data is stored on the server in encrypted form it is protected from leaking by the server (who does not know the key), and there no need to encrypt the data again when it passes on the network. Encryption is done by the original owner of the file, and updated by subsequent writers, but the server performs no encryption or decryption. The secure checksums of NASD are still needed in order to ensure the integrity of the communication, but privacy is ensured without per-byte encryption[1]. In order to use the data, the client must still decrypt it, but now using a longer-term file key that is must obtain *apriori* instead negotiated on the fly.

The problem then becomes simply one of key distribution, how users can obtain these long-term keys. This can be done via a centralized key server similar to the NASD file manager or an NIS server in today's system. The alternative uses a distributed scheme where data owners provide keys to eventual users directly. A variant of such a scheme using self-certifying pathnames is proposed in SFS [Mazieres99, Fu00]. This system is further expanded in the Cepheus file system [Fu99], with additional support for key management.

The duality between a simple optimization to secure storage communication and what is usually considered the "most paranoid" storing of encrypted data directly by users illustrates that NASD and CFS are really two points along a continuum of possible solutions, which simply trade off performance to guarantee a chosen level of security. The contention of this paper is that all the existing systems for storage security can be seen as variants on a basic set of considerations: they simply explore different points in the same solution/performance space, often without knowing they are doing so.

Section 2 outlines the axes of this solution space for a range of design options and describes the possible choices in a single, general model. Section 3 describes the behavior of a "most paranoid" system that combines the most restrictive of all of these choices - essentially the best, most secure - of all worlds. Section 4 describes how each of the systems proposed elsewhere fit into the general model, and how the choices they make reduce the security or improve the performance of the "most paranoid" scheme. Section 5 evaluates the decisions made along each of these axes using a traced workload from a UNIX time-sharing server to concretely quantify security costs in normal usage. Section 6 considers a number of issues that arise from the general model and proposes solutions that fill several gaps unaddressed by any of the existing work: key distribution from data owners, the large number of keys required in a key-per-file system such as CFS, the logical extension of untrusted servers for the "most paranoid", and the problem of space management.

[1] privacy of arguments (hiding which data is being requested by whom), to limit knowledge of access patterns, still requires encryption of requests and message headers, but file data - the vast majority of bytes transferred - would be transferred in the same, encrypted form in which it is stored.

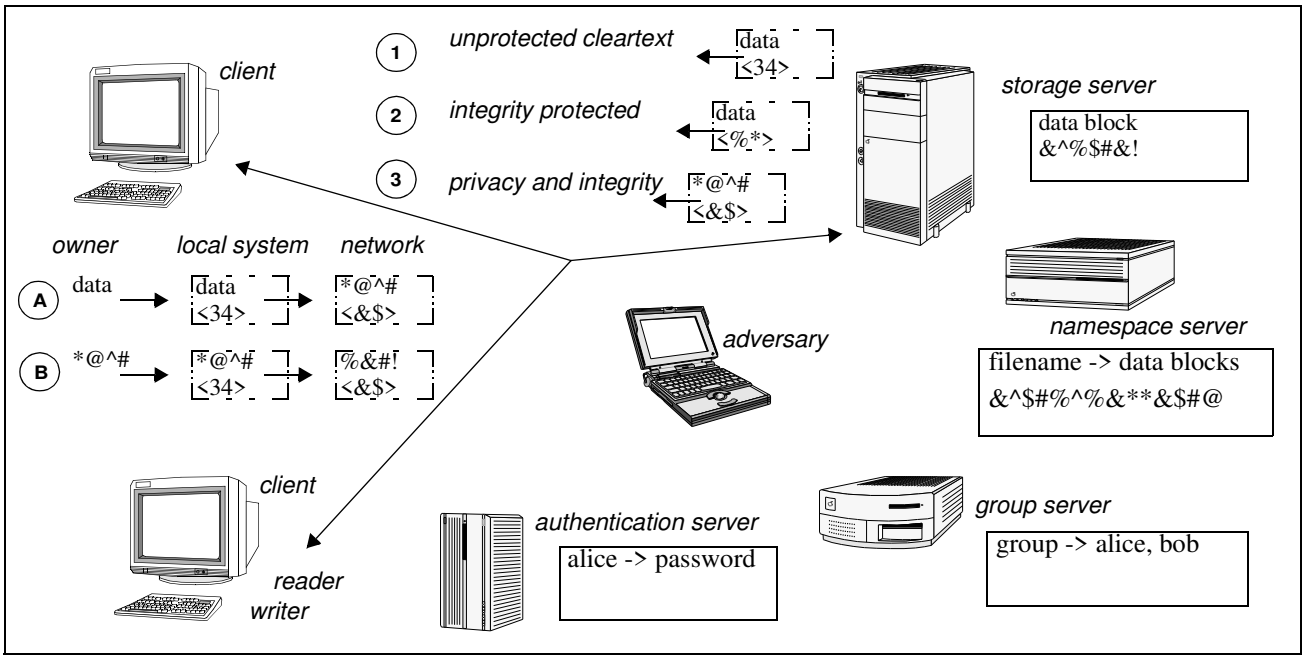

Figure 1. Diagram of all the potential players in a secure storage system. The three options for exchanging messages labelled 1 through 3 show different levels of protection from adversaries on the network - either unprotected with a simple checksum, integrity protection with an encrypted checksum, or privacy with data encryption and an encrypted checksum. The two options available to data owners are also, they can either (A) provide plaintext data to their local system which the system will potentially encrypt for transfer on the network, or (B) encrypt data before handing it to the local system which may then choose to encrypt it again for network transfer. We argue in the text that this second encryption is almost always superfluous, and allows an optimization that essentially provides "pre-computed" encryption to reduce system overhead.

2 Model of storage security

This section describes the core set of problems that must be addressed by any secure storage system, and outlines solution alternatives in a general way. The intention of this list is to condense all of the solutions used in previously proposed systems into a single framework, Section 4 will show the mapping of existing systems into this model. Figure 1 shows all of the entities described below, as well as several options for protecting data and data transfer. We start with a very simple definition:

> ***valid data*** - data created by owners as modified by writers (any other data is *invalid*)

and then add the complexity needed to protect it.

2.1 Players

The following defines the terminology we use in the rest of the paper. We believe that this list covers all of the possible players that need to be considered for protecting stored data - all other entities can be mapped to equivalence with one of these. All actions not explicitly listed for a particular player are not allowed by that player (and a secure system must prevent this), handling of items listed as *might* will depend on the user's choice of the security level they expect from their storage system.

> ***owners*** - create and destroy data, delegate read and write permission, perform revocation.

readers - read data, *might* delegate read permission.

writers - modify data, can read, can destroy (unless versioning), *might* delegate write permission, *might* delegate read permission, *might* be able to convince others that invalid data is valid.

storage servers - store data, can *always* destroy data (e.g. by storing only zeros - such destruction *might* be detectable), *might* deny updates (by not actually changing the stored data), *might* modify data (e.g. writing zeros when not requested to), *might* read data. These might be file servers as in NFS or SFS-RO, or disks as in NASD, or disk arrays as in iSCSI.

group servers - authenticate principals and authorize access based on membership groups as defined by *owners*. Owners decides who may access their data, and they *might* choose to delegate this decision to a centralized server. These could be explicit group servers as in Cepheus, NFS with NIS, and AFS [Howard88] or they may be bound with the namespace server as in NASD.

namespace servers - allow traversal of hierarchical namespaces - lookup of directories and files in directories, *might* authenticate principals to authorize traversals. These could be explicit servers separate from the storage servers, as with the file manager in NASD, or bound together as in NFS and AFS.

principals - any of the above six groups.

revoked principals - had permissions of principals at some previous point in time, but do not now.

adversaries - attempt to read, write, or destroy data when not authorized, *might* interfere with requests.

2.2 Actions

The following basic operations are defined:

creation - generate new data

read - present data to a user in cleartext

write - modify data in a way that both readers and owners will observe the change

destroy - cause data to become unreadable by readers

revocation - cause a principal to be revoked

2.3 Authentication

The purpose of authentication is to strongly establish the identity of a particular principal in order to authorize their actions against a known set of permissions for access or modification as determined by data owners.

distributed authentication - owners explicitly authenticate each principal and authorize access to the data they own (as in Blaze's CFS, or the use of server public keys in SFS).

centralized authentication - owners delegate responsibility for authentication and authorization to a group server (as accomplished through the checking done by the file server in NFS or the file manager in NASD).

The usual concern is about authentication of owners, readers, and writers to servers, but there may also be concern about authenticating *servers* to *users* to prevent improper service [Mazieres99].

2.4 Authorization

Authorization can be done in one of two general ways:

server-mediated authorization - servers receive actions and perform them on behalf of readers, writers, and owners (as in NFS and AFS).

explicit distribution of keys - owners provide readers and writers with keys that they can use to authorize or perform actions (such as the capabilities in NASD, and the server keys in SFS).

2.5 Group membership

The purpose of group membership is to compactly represent the permissions on a particular set of data by simply verifying the membership of a principal in a group, and then authorizing access based on group permissions.

distributed group membership - owners explicitly determines who is authorized to share data and distribute the necessary keys (as in Blaze's CFS).

centralized group membership - owners delegate group-based authorization to a group server that distributes keys (as in NFS w/ NIS, NASD, and Cepheus).

Access control lists are a variant of group membership that might explicitly list all the principals, but these lists must still be stored somewhere and essentially provide the group membership function [Howard88, Hughes99].

2.6 Granularity of keys

The keys used to encrypt and decrypt a particular set of data may be short-term (smaller window of vulnerability, less data encrypted with a single key) or longer-term (more efficient to manage).

session keys - last for the duration of one principal and one session (as in NASD and iSCSI or IPsec).

long-lived keys - last across sessions, *might* last across principals (as in Blaze's CFS and SFS), a critical choice is which granularity of data to associate a key with (choices include per-file, per-directory, or per-file-group).

2.7 Protecting communication

Mechanisms for ensuring reliable and secure passing of messages have been worked on for some time in networked systems and several mechanisms are popular, including SSL to protect http traffic, SSH to protect remote terminals, and IPsec to more generally protect Internet traffic [Kent98]. A variant of such a system for storage was described in the NASD work [Gobioff98]; a similar scheme is used in the self-certifying file system [Mazieres00]; and IPsec has been proposed as the primary security mechanism for iSCSI [Satran01]. Some scheme involving keyed checksums[1] will always be needed, irrespective of the design chosen, as long as storage operates in an open, networked environment. This is the only way to protect against unauthorized replay or server impersonation (man-in-the-middle) attacks.

2.8 Key distribution

In order to facilitate authorization, keys (shared secrets) must be distributed from owners to readers, writers and possibly servers. These keys include:

authentication keys - for authenticating principals (such as usernames and passwords in NFS w/ NIS, Kerberos keys in AFS, certificates in SSL, public keys in SFS-RO).

session keys - for protecting communication (such as the session keys in NASD or IPsec).

[1] the key is needed to tie the checksum to a particular principal, and the checksum is needed to tie the key to a particular set of data.

data keys - for protecting stored data (such as the file keys in Blaze's CFS).

2.9 Revocation

When a principal is revoked - e.g., a user leaves a particular workgroup - the keys to which this principal had access must be changed, and in systems where data is stored encrypted, data may have to be re-encrypted:

re-encryption - *might* have to re-write data with new key, data distributed under the old key in the past will *always* remain readable, data written under old key *might* remain readable.

lazy re-encryption - *might* do only future updates with new key [Fu99].

periodic re-encryption - change of keys *might* be done periodically- perhaps nightly or weekly - to limit the window of vulnerability [Gobioff99a].

2.10 Versioning

The only way to completely prevent destruction of data by modification or overwrites is to maintain data pages as copy-on-write or use logging, while never deleting old versions of data, as in self-securing storage[Strunk00].

2.11 Space Management

The allocation and deallocation of storage blocks.

denial of service - an attack *might* be possible where adversaries can write useless data and fill the storage device.

garbage collection - the difference in design arises in who has the ability to traverse the directory and file structures to reclaim this space.

3 Most paranoid storage system

We now outline a system that provides a fully secure storage system with a superset of the properties discussed above, making the most restrictive, or "paranoid" decisions at each point (we realize the design will seem redundant as described, please bear with us, and we will clear up the redundancies in the next section).

The lifetime of a single file in a "most paranoid" storage system:

(1) a file *secret-notes* is created by user *wilkes* and encrypted with a secret key WK_1*(secret-notes,wilkes)* chosen specifically for this file (this file creation and encryption might be repeated 1,000 times over the course of a day)

(2) the encrypted file is transferred to a shared server *cello* and stored - the communication between *wilkes* and *cello* is protected by a session key SK_i*(wilkes,cello)* generated for purposes of this single *write* request (the generation of a session key between *wilkes* and *cello* will be repeated 100,000 times during the day, *cello* may perform up to 5,000 such operations per second during the busiest time of the day)

(3) the write of the file is authorized by *wilkes* authenticating to *cello* using his password *PK(wilkes, cello)* and *cello* checking the mode bits of the parent directory */home/wilkes* to verify that *wilkes* has write access

(4) user *wilkes* now tells his colleague *alice* about the file (repeated 3,000 times during the day)

(5) user *alice* retrieves the encrypted file from *cello* - the communication between *cello* and *alice* is again protected by a session key SK_i*(alice,cello)* generated for purposes of this single *read* request (the generation of a session key between *alice* and *cello* might be repeated 40,000 times over a day)

(6) the read of the file is authorized by *alice* authenticating to *cello* using her password *PK(alice, cello)* and *cello* checking the mode bits of the file to verify that *alice* has read access

(7) to decrypt the file, *alice* must obtain the read key, RK_1*(secret-notes, wilkes)* corresponding to the write key that encrypted the file (*wilkes* might have to distribute such keys for reading 6,000 times, and alice might obtain 4,000 such read keys over a day)

(8) if *alice* now wants to modify a few lines of the file, she can make the change in her local plaintext copy, but must obtain WK_1*(secret-notes, wilkes)* in order to encrypt the file for storage and future decryption (this modification of files owned by another user might occur 300 times over the day)

(9) this encrypted file is transferred to the shared server *cello* by *alice* and stored, the communication between *alice* and *cello* is again protected by a session key SK_{i+1}*(alice,cello)* generated for purposes of this single *write* request

(10) the write of the file is authorized by *alice* authenticating to *cello* again using her password *PK(alice, cello)* and *cello* checking the mode bits to verify that *alice* has write access

(11) user *cathy* is also interested in reading the file, so she obtains the key RK_1*(secret_notes, wilkes)* as well

(12) at some future point, *alice* receives a better offer from a startup and leaves the company - since she still knows the keys RK_1*(secret-notes, wilkes)* and WK_1*(secret-notes, wilkes)*, the file must be re-encrypted with a new key WK_{i+1}*(secret-notes, wilkes)* and the corresponding read key must be distributed to *cathy* (this revocation and re-encrypting may occur once every few months in a medium-sized user population, and could affect several thousand files and gigabytes of stored data)

Figure 2. Scenario for file creation and access in a "most paranoid" storage system. The steps required to handle operations on a sample file, with each step providing an idea of how often each operation might happen over the course of a day in a real storage system, based on the trace data introduced in the next section.

The description in Figure 2 follows the lifetime of a single file through a "most paranoid" storage system. All of the numbers mentioned are example per-user values from a 10 day file system trace taken on a UNIX server with about 25 active users and close to 500 gigabytes of data in over 10 million files (see Section 5).

4 Less paranoid storage systems

The first widely-known discussion of security for storage systems is the Cryptographic File System (CFS) by Matt Blaze of Bell-Labs [Blaze93]. He proposes a system that uses a secret key to protect a directory in a file system. The underlying data is stored as a single file in the host file system, and "attached" as a cleartext directory under a */crypto* mount point. This allows the host file system to treat the encrypted data as just another file, so normal utilities such as backups work unchanged, but do not have access to the cleartext data. The system is implemented as a user-level NFS loopback server. The key characteristics of this system, using the terminology introduced above are:

- owners, readers and writers are indistinguishable, the storage server is the host file system, group membership is handled by owners, namespace traversal is handled by readers
- owners handle authentication by distributing keys to encrypted directories and files
- owners do group membership by distributing keys
- authorization is done by passing keys to readers and writers
- long-lived keys are used on a per-directory basis
- no protection of communication is done (this is essentially a local system)
- authentication is possible only by having the key, one key protects everything

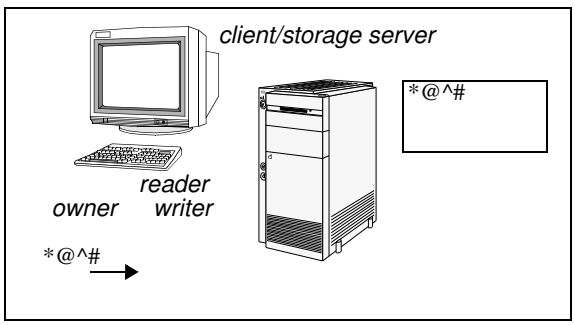

Figure 2. Diagram of Blaze's Cryptographic File System. This is essentially a local system, with client and storage server on a single machine. Namespace service and group membership are both managed directly by owners distributing keys. Owners pass data to the system in encrypted form, and that is exactly how it is stored.

- trusts readers and writers, but not the storage server
- revocation requires re-encrypting all data

As an additional consideration, a later paper by Blaze [Blaze94] introduces a key escrow system to recover keys after an owner has left the organization.

The CryptFS system from Columbia [Zadok98] extends the Blaze system to:

- use process and user session keys, rather than usernames (this expands owners to be individual *user sessions*, rather than users as in CFS)
- be more efficient as it is built as a stackable file system rather than a user level server

The Truffles system from UCLA [Reiher93] uses an alternative method of key-management to handle the problem of generating keys when owners are not available by splitting keys such that any *n* members of a group can collude to generate the key of a missing owner.

All of the above systems assume untrusted servers, i.e. keys are known only to the owners and readers, and not trusted to the system itself. The key escrow system of Blaze depends on trust of the key database, but not trust of the storage servers. The Truffles system distributes this trust among a group of owners.

A number of systems - including AFS and DFS - address the problem of untrusted clients by using a system such as Kerberos to authenticate users to servers [Neuman94]. A system such as this, or a similar one relying on certificate authorities is required by any storage system in order to strongly authenticate users.

The work of Mazieres at MIT addresses the problem of securely authenticating *servers* to *readers* by:

- associating a public key with a set of files to allow a reader to verify that a particular set of data were those created by the original owner of the corresponding private key
- adding these public keys to file pathnames in order to easily distribute them
- using a central database of revocation lists to track revoked keys

in terms of the concepts introduced above:

- there are only owners and readers, no writers
- authentication of readers is not done, namespace traversal is done by the readers who know the server name and key
- group membership is also done by knowing the server public key and name
- authorization is done by key distribution
- granularity of keys is per-file-system
- communication is protected via the server key
- key distribution is via self-certifying pathnames

- servers or owners do not trust readers, but readers can verify servers
- revocation of servers requires readers to check for revocation of any key before use

This allows the wide sharing of read-only files, even when servers are untrusted.

The Cepheus file system uses this infrastructure to add:

- sharing of files by associating a user key and group key with each file
- delayed re-encryption to make revocation of keys less costly

[more description needed here, lock boxes, etc.]

[need a section for the system described in Hughes99]

The work of Gobioff on Network-Attached Secure Disks (NASD) proposes a system that:

- handles key distribution by having file managers distribute capabilities to untrusted clients for particular storage objects
- handles authorization by having clients present these capabilities to servers
- using checksums and message authentication codes (MACs) to ensure the integrity of requests and data transfer to/from the servers
- encryption may be used to ensure the privacy of request and data transfer to/from storage servers

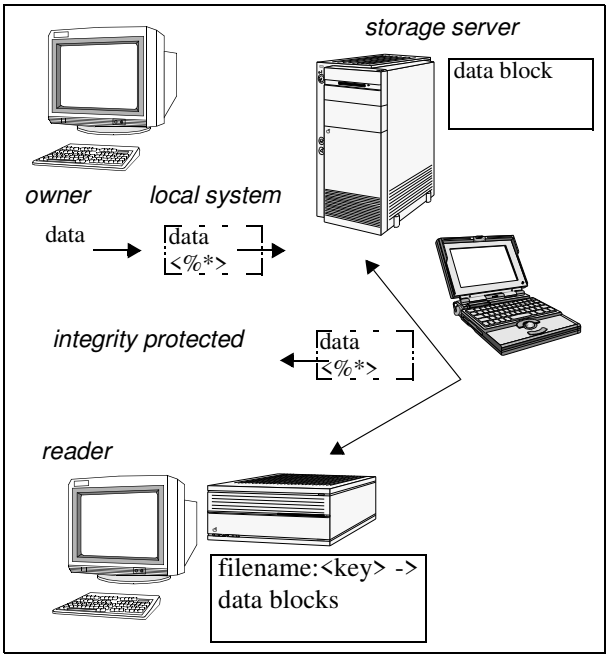

Figure 3. Diagram of Self-Certifying File System. Owners store data with secure checksums, checksums are distributed using the server's public key, allowing clients to verify that data came from the server. Namespace traversal is done by the reader.

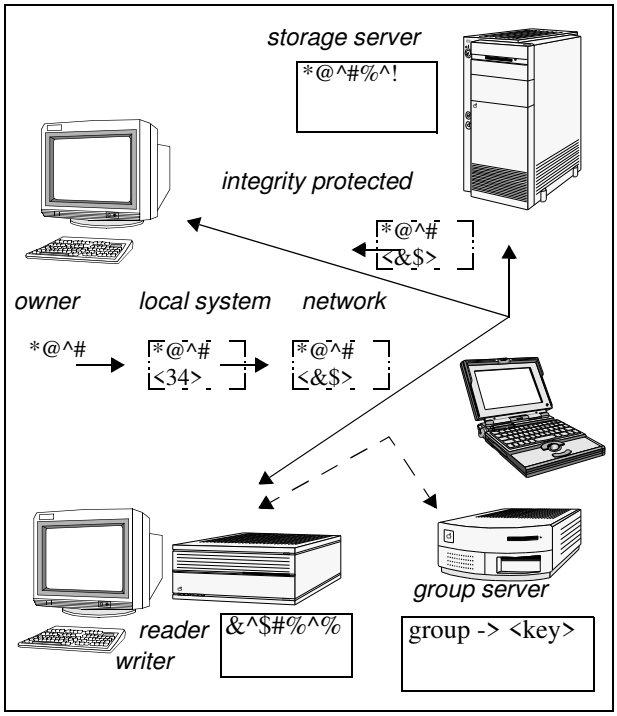

Figure 4. Diagram of Cepheus File System. Data is encrypted by owners before it is stored. Data transfer is protected by secure checksums. Namespace traversal is done by clients, and key distribution is handled by a centralized group server using lock boxes for revocation.

- introduces a scheme of pre-computed checksums to reduce the computation of generating checksums on each individual request

Finally, the recently proposed iSCSI standard for networked storage [Satran01] suggests:

- the use of IPsec to protect communication between clients and storage servers
- negotiation of session keys on a per-login basis
- key distribution by an external mechanism for user authentiation

[more description needed here for iSCSI and IPsec, also consider the Klein proposal]

[also consider - Secure NFS, more details on AFS]

Two research projects currently underway attempt to address data security and long-term protection on a much wider scale - Oceanstore is considering a world-wide set of encrypted replicas of user's data [Kubiatowicz00] and PASIS is considering a similar direction, but very different approach, with survivable storage where data integrity must be maintained in the face of loss or destruction of some number of replicas [Wylie00].

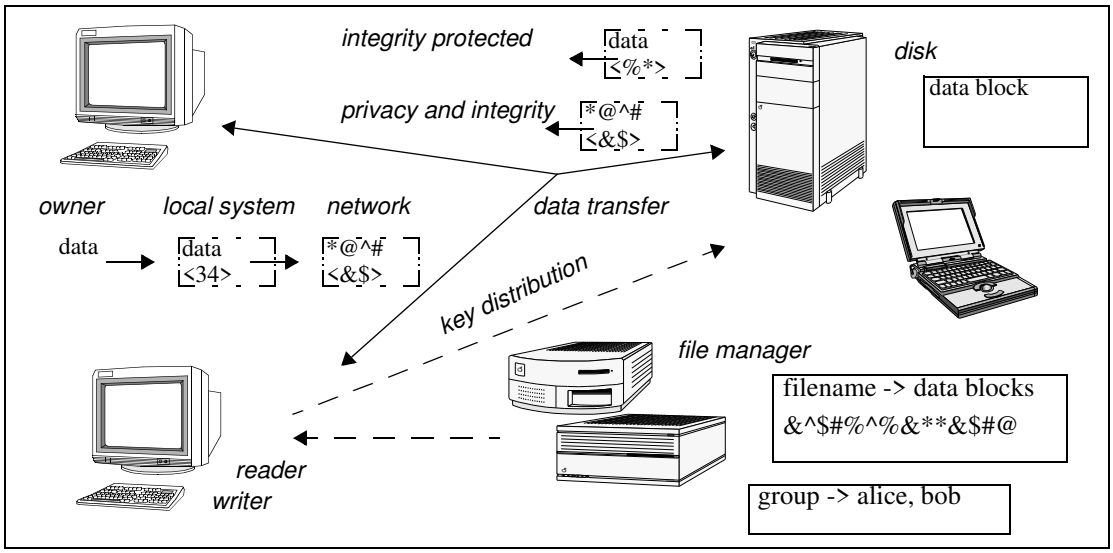

Figure 5. Diagram of Network-Attached Secure Disks. Two options for data protection: integrity which protects against modification in transit, and privacy which ensures data is not seen in transit. Data is presented as plaintext by owners and stored in plaintext on the disk. Namespace management and group membership are handled by a file manager separate from the disks via distribution of long-term capabilities (keys).

5 Evaluation

This section explores the cost of implementing the various design choices discussed above. The context for evaluation is a 10-day trace of all file system accesses done by about 20 users of a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage space[1]. The trace was collected by instrumenting the kernel intercept and logging all file system calls at the syscall interface. Since this is above the file buffer cache, it means that the numbers shown will be pessimistic to a system that attempts to optimize server messages or key usage based on repeated access. A later section attempts to quantify this effect for a selected set of files. Table 1 provides an overview of the trace.

	12-hour	*10-day*
hours	12	240
requests	11.5 million	?
data moved	23 GB	?
active users	23	32
user accounts	207	207
active files	111,000	?
total files	3.6 million	3.6 million
file systems	24	24

Table 1. *Overview of file system trace.*

5.1 Dimensions of Comparison

The purpose of presenting this data is to compare the relative costs of the systems discussed in the previous sections using numbers from a realistic system. This allows us to see how often expensive operations such as full-bandwidth encryption, key distribution, or key generation would occur in practice. We consider a number of these metrics in turn.

5.2 Session vs. per-file keys - readers & writers

Table 2 gives counts for the total number of keys used in each of the two high-level classes of designs - using session keys or long-term, per-file keys. The table shows the number of keys on a per-user basis for several representative and several system userids during the trace period.

Table 3 considers a per-group key scheme, showing the number of keys each user would need to obtain during the trace period if keys were created only for each "permission group" of files - i.e. where all files that have the same owner, group, and permissions bits share a single key. Note that the number of keys required is orders of magnitude lower than in the per-file scheme.

[1] due to time constraints, we only analyze a subset of the total trace for this paper, so all further numbers in the paper will refer to the 12-hour trace (8am to 8pm on a single day). The entire 10-day trace will be complete in time for the final paper, and will also give enough data to allow study of a revocation scenario.

	session keys			per-file keys			
user	per request	per open/close	per filesys/lv	total	owner	non-owner	newly created
wilkes	15,600	1,632	8	251	25	226	13
alice	113,400	10,705	15	2,145	1,290	855	1,073
bob	11,300	1,410	4	272	83	189	64
root	6,590,000	393,000	17	11,728	3,350	8,378	7,761
news	1,667,000	264,000	3	103,218	103,176	42	79,248

Table 2. *Per-user dynamic metrics - number of keys used.* The number of keys needed if encryption is done on a per-session basis (using three different definitions for session - a session per request, a session per open/close pair, and a single session per file system or logical volume) as in NASD, SFS, or iSCSI vs. on a per-file basis (as in CFS). Total number of per-file keys by username is separated into the total keys used, number of those keys owned by the user, the number that would have to be obtained from another owner, and the number of new keys created.

	per-group keys			
user	total	owner	non-owner	newly created
wilkes	28	4	24	0
alice	55	6	49	0
bob	34	7	27	0
root	233	30	213	0
news	13	6	7	0

Table 3. *Per-group dynamic metrics - number of keys used.* The number of keys that a user needs if per-group keys are used instead of per-file keys. No new per-group keys were created during the course of the 12-hour trace.

5.3 Per-file vs. per-group keys - owners

Considering the complexity for owners, as opposed to readers and writers, Table 4 looks at the number of keys that would have to be managed by data owners in a key-per-file system. The table shows the total number of keys needed by owner. The static column gives a count of all the files in the entire file system owned by the given owner. The dynamic numbers show the number of keys the given owner would have had to distribute in the time of the trace, both to readers and to writers. We can see from these numbers that a system requiring direct user involvement for key distribution such as Blaze's CFS would be prohibitively cumbersome (imagine typing in 4,000 keys - from a possible list of 200,000 - in several hours at your desk).

per-file	static	dynamic	
user	files owned	read keys distributed	write keys distributed
wilkes	54,520	274	8
alice	19,413	375	218
bob	216,462	4,381	32
(top 10 users)	84,479	834	36
(all users)	5,508	49	3
bin	191,466	6,331	19
root	240,366	2,170	54
news	1,574,260	19	2

Table 4. *Per-user static metrics for per-file keys.* Assuming a system that uses per-file keys, how many keys must a particular owner be aware of. The static column shows the totals for all the files that exist in the file system, and the dynamic column shows the number distributed during the 12-hour trace.

per-group	static	dynamic	
user	groups owned	read keys distributed	write keys distributed
wilkes	28	14	1
alice	13	6	4
bob	17	13	3
(top 10 users)	23	10	3
(all users)	7	3	1
bin	33	28	3
root	129	34	7
news	15	8	1

Table 5. *Per-group key static optimization.* The number of keys that a user needs to handle if per-group keys are used (because of files that the user owns).

Table 5 shows the change in number of keys required by individual owners if we move to a key-per-file-group scheme. In this case, there is not a separate key for each file, but a key for each class of files, as in the previous section. This produces a much more manageable list with roughly 20 keys per owner, and 10 of them distributed during a 12 hour period. Note that these numbers are also skewed high since we assume the readers and writers do not already have any keys cached when the trace starts. In practice, or in a longer trace, the number of keys to be distributed each day would be even lower.

		number of ops		NASD	CFS	SFS	Cepheus
	operation	messages (000s)	bytes (MB)				
integrity	MACs	11,588	-	X	-	X	X
	checksums	7,814	82,543	X	-	X	-
	pre-computed checksums	7,814	11,655	optional	-	-	X
session privacy	encryptions - server	4,868	40,930	X	-	-	-
	decryptions - server	2,946	41,613	X	-	-	-
file privacy	decryptions - client (avg)	212	1,780	X	X	optional	X
	encryption - client (avg)	128	1,809	X	X	optional	X

Table 7. *Number of cryptographic operations for each design.* The number of cryptographic operations performed by the server, and the average client (user). The key cost difference between the systems can be seen in the resources required for session privacy. Integrity must be provided in one form or another whenever adversaries might intercept or modify requests in transit. The choice between session privacy and file privacy comes down to performance cost vs. simplicity of centralized management. With appropriate key distribution schemes, file privacy can provide a much more computationally efficient means of achieving the same level of security.

5.4 Runtime cost - design comparison

The total number of operations and amount of data transferred by several users and the two busiest system userids, as well as the system-wide total for the entire 12-hour trace is shown in Table 6.

	requests (000s)			data bytes	
user	total	read	write	read	write
wilkes	16	4	0.2	8 MB	450 KB
alice	113	54	8	149 MB	28 MB
bob	11	4	0.7	8 MB	12 MB
root	6,590	2,999	1,522	9 GB	15 GB
news	1,667	274	614	202 MB	19 GB
total	11,588	4,868	2,946	41 GB	42 GB

Table 6. *Per-user and total requests and data moved.* The number of messages and number of data bytes moved by the server (total) and by several individual users in our trace.

Table 7 shows the total number of cryptographic operations required to provide different levels of security. The table also shows which of the systems introduced in the previous section incur which of these costs.

6 Potential optimizations

This section summarizes a number of potential optimizations that arise in the model and from the data presented in the previous section. These should be considered extensions to the systems already described in Section 4.

6.1 Distributing owner keys

A logical extension to the key distribution scheme used in Cepheus that would make it closer to the purely owner-managed scheme of CFS is to distribute key and group management directly to data owners, as shown in Figure 6. In order to obtain a read key for a particular file, a reader sends a message to the file owner requesting the key. The owner returns the key to the reader if he is authorized - perhaps consulting a group server to determine group access, or perhaps authorizing the reader directly. As shown in Table 5, the number of keys that the average user would have to distribute in a day under such a scheme would be quite low, only 15 for the busiest owner, and 40 for the root owner (which would likely be handled as an automatic server). For the paranoid, this means that owners could interactively authorize the keys sent to readers and writers - without having to trust a software agent even on their local system. Alternatively, owners might use external systems such as smart cards to manage access to their keys [Hughes99].

6.2 Sharing via public-private key pairs

The same file data can either be read or written, depending on authorization, so a mechanism is required to separately authorize read and write to the same encrypted data. Public-key cryptography can be used to design a simple scheme that allows shared access while differentiating between the capabilities of readers and writers.

The private key can be used as a *write key* to encrypt the file during a write and the public key can be used as a *read key* to decrypt the file for reading. The keys themselves are generated and maintained by the owner, as described above. With such a key pair, handing a principal the read key authorizes that reader to read the data, and handing a principal the write key authorizes that writer to modify the file in a way that allows other readers to successfully read the changed data.

Though the read key is equivalent to the public key in a standard public-key system, it is *not* publicly disseminated. The owner of the file will issue this read key only to what it considers as authorized readers. Similarly, the write key is handed to writers only after appropriate

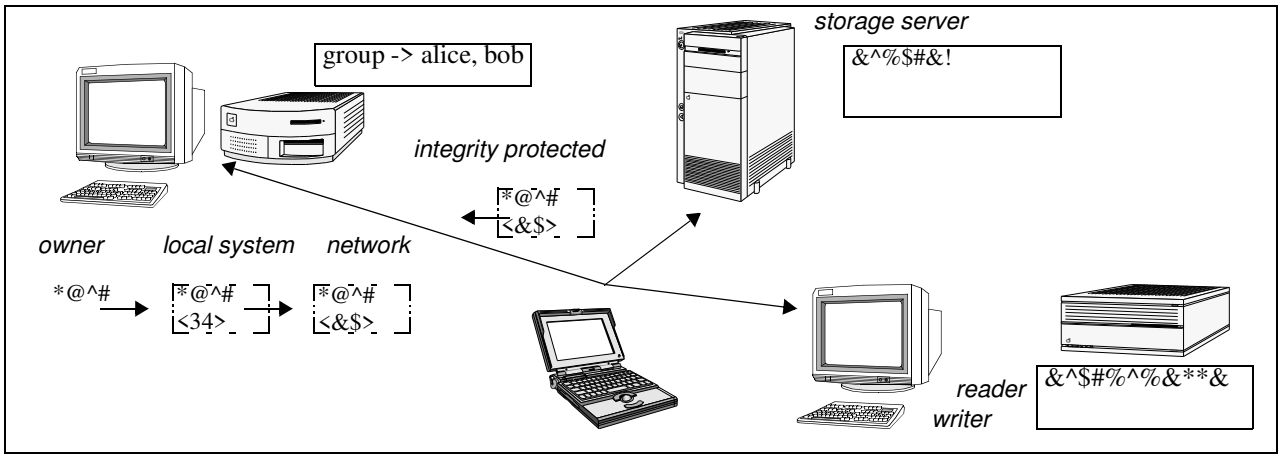

Figure 6. Diagram of owner-managed key scheme. The owner writes the data in encrypted form, and it is stored in encrypted form. Namespace management is done by readers and writes, and directories are also stored encrypted. Key distribution is handled directly by data owners, rather than by a centralized group server. Each owner is responsible for distributing read and write keys when queried.

authorization by the owner. In the basic variant of this design, there is no protection from a malicious writer who changes the write key used to encrypt the file: the readers and the owner would not be able to decrypt this newly written file and would read only garbage.

6.3 Precomputed encryption

Storing files encrypted has the obvious benefit of securing the contents from unauthorized access. In addition, it reduces the load on the server which would otherwise have to perform encryption on-the-fly to protect data during network transfer. This can be quantified by comparing the amount of encryption and decryption done by the server in NASD and the Cepheus file system as shown in Table 7.

On the downside, this makes the write key of the file a long-term key. The difficulty with using a long-term key is that much larger amounts of data are encrypted with the same key and can be seen on the network, making it more vulnerable to snooping and attacks that depend on large amounts of available cypertext. This risk may be reduced if each block of plaintext is extended with some amount of random padding before encryption. It may also be reduced by periodically refreshing the keys used, and potentially re-encrypting the underlying data (perhaps using a lazy scheme, as discussed below). This would incur an additional encryption cost, but would still be less costly than re-encrypting the file contents with a session key each time they are sent to a reader.

6.4 Reducing the number of keys - file-groups

Pre-computed encryption together with owner-managed key distribution implies that the owner must maintain and distribute the keys to all their files. The most paranoid design would require one key per file that the owner owns in the system. This would result in an manageably large number of keys for the owner to handle, as shown in Table 4.

The number of keys can be reduced dramatically while still maintaining the semantics of current UNIX file sharing by observing that if two files are owned by the same owner, the same group, and have the same permission bits, then they are authorized for access by the same set of users. We place all such files in the same *file group*, and use the same key to encrypt all the files in that group.

Using file groups dramatically reduces the number of keys that readers and writers need to keep track of (as seen in Table 2 and Table 3 in the previous section), as well as the number that owners are required to manage (as shown in Table 4 and Table 5). Additionally, this scheme reduces the number of requests for keys that an owner must process, and the number of keys that a principal must obtain from a remote owner. When a principal needs to access a file it can check if the file's ownership and permissions match that of any other file for which it already has a key, and simply use the cached key.

Again, using the same key to encrypt multiple files has the disadvantage of making more data that has been encrypted with the same key visible on the network, as mentioned in the previous section.

Additionally, using the same key for different files makes revoking a principal an expensive operation: all files in the file group will need to be re-encrypted with a new key. If a different key were used to encrypt each file, then only the files to which a revoked user had access are required to be changed (although this would require logging of all key distributions, as it would degenerate to the same cost as the group-key scheme if all files the principal *potentially* had access to had to be re-encrypted).

6.5 Lazy re-encryption

Using file-groups drastically reduces the number of keys in the system, but complicates revocation. To make revocation less expensive, one can perform re-encryption only when a file has been updated. In this way, expensive re-encryption occurs only when new data is created, the concept of *lazy re-encryption* [Fu99].

When a principal is revoked, other principals must be informed to begin using a new key. The protocol to inform readers is different from that used to inform writers. Readers can notice that a file has been re-encrypted when the checksum of the decrypted disk block does not match the checksum stored in the block. If the check fails the reader can request a new key from the owner.

If writer's keys were renewed periodically (e.g. once per day), then writers could be informed that they now need to use a new key. The first writer to update the file also re-encrypts it with the new key. This opens a window of vulnerability between revocation and re-encryption. If a higher level of security requires revocations to take effect immediately, the owner could proactively perform the re-encryptions and writers would detect key changes in the same way as readers.

Since revocations occur infrequently, the expense of re-encrypting all the files to which the revoked principal could have had access is justified by the reduction in the number of keys in normal usage. A revoked principal who has access to the server will still have access to the files which haven't been updated since the principal's revocation. This may be acceptable since it is equivalent to the access the principal had during the time that they were authorized (when they could have been copying the data to floppy disks, for example). No matter what, revoked principals will never be able to read data created since their revocation.

[this revocation must be better explained earlier on - in the basic model section - so that we use the same terminology. better explain what types of scenarios we are worried about]

6.6 Central authorization with untrusted servers

The logical extreme to not trusting storage servers to read the data they store, is not to trust servers with namespace traversal or write authorization. Readers must be able to get a listing of all the filenames in a directory in order to traverse the namespace. At the same time, we would like to prevent writers from destroying the contents of a file (e.g. by writing with an invalid write key). Since writes make changes to persistent data, the server must authenticate each write to make sure the writer is authorized to modify that particular data block.

6.6.1 Cascaded keys

Instead of using a heavyweight authentication protocol on each write, we simply require the writer to provide the encrypted filename of the file being updated, encrypted using the *write authorization key* of that file. The writer obtains this key from the owner together with the usual write key. Since the write authorization key is secret, the encrypted filename serves as a certificate to the server that the writer has the write authorization key. The server checks if the encryption is valid by searching the directory for the encrypted filename before proceeding with the write. The server can do this comparison without itself knowing the key used by the writer.

6.6.2 Directory management

We propose a simple scheme, illustrated in Figure 7 based on encrypting the filenames with two keys. The directory file contains the information required by the server. The directory contains a table of names of the files in the directory, each encrypted first with the write authorization key and then with the *write verification key*. The directory does not contain the plaintext filenames at all, preventing an unauthorized user from even performing a directory lookup. The directory is maintained such that the index of a filename in the encrypted table matches the index of the data structures associated with that file. The directory file itself is not encrypted; that is, the server can access the block addresses of files in the directory. The server is also provided with the write verification key so that it can verify the write authorization certificates presented by the writers.

To allow a reader to access the names of files in the directory, the owner gives readers a key which corresponds to the product of the write authorization key and the write verification key. The reader can use this key to completely decrypt the filenames and obtain the plaintext names. However the reader cannot obtain a valid write

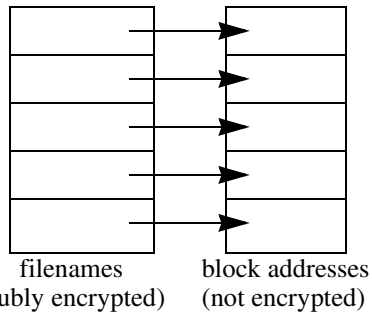

Figure 7. Diagram of directory storage on untrusted servers. The right-hand table stores pointers to the data blocks corresponding to the named file. The left-hand table is encrypted in such a way that it can be decrypted to plaintext by readers, searched by the storage server on behalf of writers, but written only by writers.

certificate from the filename because it cannot factor the key obtained from the owner into the two constituent keys. This scheme allow readers to cache the file data block addresses to use in future accesses. On receiving a certificate from the writer, the server decrypts the doubly encrypted filenames using the write-verification key and verifies that the certificate provided by the writer corresponds to a valid entry. The server can cache the decrypted filenames of active directories to make this validation fast.

The critical point is that this scheme uses the encrypted filename to indicate to the server that the writer is authorized. Hence it is important that this certificate be protected during transmission, using the privacy of arguments scheme as proposed by NASD [Gobioff99]. Such a design allows an untrusted server to verify that a principal has the required authorization, while mediating writes to protect data from malicious modification. The scheme also makes it easy for the server to manage the space on the storage system by decoupling the information required to determine allocated space from the data itself. Though the actual data and filenames are encrypted and hidden from the server, the list of physical blocks allocated is visible to the server for allocation decisions.

6.6.3 Alternate design

For each file in a directory, along with the (plain text) table of filenames, the directory contains an additional table corresponding to the filename encrypted with a write authorization key. The entire directory file is then treated as any other file: it is encrypted with a write key. To allow readers to perforn name lookup, they are handed the read-key to the This scheme is illustrates in Figure 8.

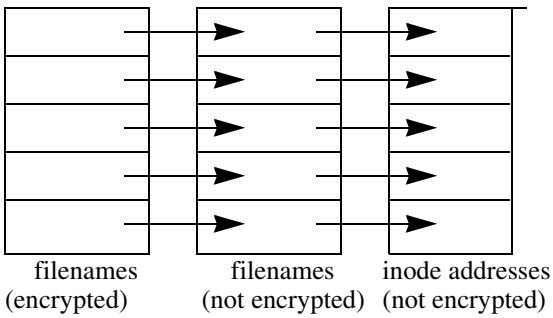

Figure 8. Diagram of alternate directory storage. The right-hand table stores pointers to the data blocks corresponding to the named file. The left-hand table is encrypted in such a way that it can be decrypted to plaintext by readers, and the center table is encrypted so that it can be searched by the storage server on behalf of writers, but written only by writers.

There is no reason why the private key of a directory needs be handed out, unless you want to authorize a user to create new files in that directory. In this case the authorized user will be able to delete *any* file from the directory as well.

6.6.4 Who knows what

wilkes: *PrK(notes), PuK(notes), PrK(notes-dir), PuK(notes-dir)*

alice: *PuK(notes), E[PrK(notes-dir)](notes)*

bob: *PrK(notes), E[PrK(notes-dir)](notes)*

cello (server): *E[PrK(notes-dir)](notes), E[PrK(notes)](notes), E[PrK(notes-dir)](notes-dir)*

6.6.5 Potential Attacks

This scheme is vulnerable to the following denial of service attacks:

- the server can delete files (e.g. by zeroing a disk)
- an anonymous user can cause the disk to be unusually loaded by requesting arbitrary data blocks

There is no reason why the private key of a directory needs be handed out, unless one wants to authorize a user to create new files in that directory. In this case the authorized user will be able to delete *any* file from the directory as well. A *malicious attacker*:

- cannot decrypt (read) a file because he does not have the corresponding public key.
- cannot forge a file because he does not have the corresponding private key.
- cannot overwrite the file *notes* because he does not possess the encryption of notes.

Similarly, a *revoked reader*:

- cannot decrypt the file because the encrypting key has been changed.

Finally, a *revoked writer*:

- cannot read the file because the encrypting key has been changed.
- cannot overwrite/forge the file because the encrypting key of the directory has been changed.

7 Putting it all together - file ops

This section combines all of the most paranoid choices in the schemes discussed above and presents the protocols used to perform the basic file system operations. Mutual authentication is required before undertaking any of the following operations, and a session key is exchanged as a result of the authentication. All messages (other than the file data) are encrypted with this session key for privacy, and include a checksum for integrity during transmission. The server must not perform any authentication

prior to reading data blocks. The data is secured using long-term encryption and pre-computed checksums.

7.1 Directory lookup

To traverse the namespace, a user must have keys to decrypt the entries in the directory. As described in Section Section 6.1, this key is obtained from the owner of the directory and allows the user to determine the names of all files in that directory

7.2 Reading a file

When a user, *bob*, needs to read a file, he must first obtain the read key for that file. He determines the owner of the file from the directory entry, and obtains the read key after authenticating himself with the owner. Subsequently reads proceed with *bob* requesting particular data blocks and decrypting them with the read key.

7.3 Updating a file

Two keys are required to update an already existing file: the write key and the write authorization key. The write key is the key with which the file has been encrypted. The write authorization key is used by the writer to prove to the server that the writer has been authorized to write that file. The writer, *alice*, obtains these keys from the owner of the file and the owner of the directory respectively. She then presents the name of the file encrypted with the write authorization key to the server as a certificate to prove authorization. The actual file data is sent after encrypting it with the write key. After checking the certificate, the server proceeds with the write as described in Section 6.6.

7.4 Updating a directory / Creating a new file

To create a new file, the user must have authorization to write to the directory *dir* in which the file *notes* is to be created. Note that the authorization to write to a directory recursively implies the authorization to update the contents of the directory above.

The writer request the directory *dir*, which is stored encrypted in the server. On receiving it, the writer decrypts it with the key, adds an entry with the name *notes* and data block number *null*. He then re-encrypts the filename table in *dir* with the same key and sends it back to the server together with the encrypted filename.

The server uses this encrypted filename to determine the index of *notes* in *dir*, and fill in the data block value corresponding to *notes*. After this, there will be an entry in the directory *dir* correctly pointing to an empty *notes* file.

7.5 Deleting a file from a directory

The writer requests the directory *dir* from server. On receiving it, the writer decrypts it with the read key, removes the entry *notes* and adjust the other data block pointers. It then re-encrypts the filename table in *dir* with the same key and stores it back to server.

7.6 Revoking permissions

Revoking readers is straightforward: keys must be renewed every day.

On the other hand, revoking writers is more involved. The owner generates two new public-key and private-key pairs. It then, re-encrypts *notes* with a new private key and store it. Finally, it re-encrypt the directory *dir* with a new private key and stores it. The owner returns the new keys to any readers or writers who request keys as their previous keys expire.

8 Conclusions

This paper has developed a common model and set of design considerations required for any secure storage system. We have reviewed all the previously proposed systems for storage security, and mapped them into this core set of components and design choices. For integrity, any system for secure, networked storage must provide some variant of signed checksums that strongly tie particular data to particular principals. For privacy through encryption, we have shown that the two main classes of systems previously described - 1) those that seek solely to protect the communication between servers and users and 2) those that allow for encryption and decryption only at user endpoints, with untrusted servers in between - are actually very similar. The second systems can be seen as providing a form of “pre-computed encryption” for optimizing the encryption work done by the servers in the first class of systems. We have quantified the costs of these various systems using a trace from a UNIX time-sharing server and shown that significant optimizations are possible that both reduce complexity and reduce server encryption load - sometimes by orders of magnitude. We have identified and begun to quantify a number of additional design choices that improve security and performance: fully owner-based key distribution, key pairs for read and write, precomputed encryption, file groups, lazy re-encryption, and authorization and space management with completely untrusted storage servers.

References

[Blaze93] M. Blaze. A cryptographic file system for UNIX. *Proceedings of 1st ACM Conferenc on Communications and Computing Security*, 1993.

[Blaze94] M. Blaze, Key management in an encrypting file system, *Summer USENIX*, June 1994.

[Cattaneo97] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Cozzolino, E. Mauriello and R. Pisapia. Design and implementation of a transparent cryptographic file system for UNIX. *Technical Report, University of Salerno*, 1997.

[Fu99] K. Fu. Group sharng and random access in cryptographic storage file systems. *MIT Master's Thesis*, June 1999.

[Fu00] K. Fu, M. Kaashoek and D. Mazieres. Fast and secure distributed read-only file system. *OSDI*, October 2000.

[Gobioff99] H. Gobioff, D. Nagle and G. Gibson. Embedded Security for Network-Attached Storage. *Technical Report CMU-CS-99-154*, June 1999.

[Gobioff98] H. Gobioff, D. Nagle and G. Gibson. Integrity and Performance in Network Attached Storage. *Technical Report CMU-CS-98-182*, December 1998.

[Gobioff99a] H. Gobioff. Security for high performance commodity subsystem. *Ph.D. Thesis, CMU-CS-99-160*, July 1999.

[Howard88] J. Howard, et al. Scale and performance in a distributed file system. *ACM TOCS* 6 (1), February 1988.

[Hughes99] J. Hughes and D. Corcoran. A universal access, smart-card-based, secure file system. *Atlanta Linux Showcase*, October 1999.

[Kent98] S. Kent and R. Atkinson, Security Architecture for the Internet Protocol, *RFC 2401*, November 1998.

[Kubiatowicz00] J. Kubiatowicz, et al. OceanStore: An Architecture for Global-Scale Persistent Storage. *ASPLOS,* December 2000.

[Mazieres99] D. Mazieres, M. Kaminsky, M. Kaashoek and E. Witchel. Separating key management from file system security. *SOSP*, December 1999.

[Neuman94] B. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications* 32 (9), September 1994.

[Reiher93] P. Reiher, J. Cook and S. Crocker. Truffles - A secure service for widespread file sharing. *PSRG Workshop on Network and Distributed System Security*, 1993.

[Satran01] J. Satran, et al. iSCSI draft standard. *www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-03.txt*, January 2001.

[Strunk00] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules and G. Ganger Self-securing storage: protecting data in compromised systems. *OSDI*, October 2000.

[Wylie00] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote and P. Khosla. Survivable information storage systems. *IEEE Computer*, August 2000.

[Zadok98] E. Zadok, I. Badulescu and A. Shender. Cryptfs: A stackable vnode level encryption file system. *Technical Report CUCS-021-98*, 1998.