Appears in the *Proceedings of the 1st Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

A framework for evaluating storage system security

Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan
Hewlett-Packard Laboratories
Palo Alto, California
{riedel,maheshk,swaram}@hpl.hp.com

Abstract

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 security desired by data owners, and the level of inconvenience users are willing to tolerate. Current storage systems secure data either by encrypting data on the wire, or by encrypting data on the disk. These systems seem very different, and currently there are no common parameters for comparing them. In this paper we propose a framework in which both types of systems can be evaluated along the security and performance axes. In particular, we show that all of the existing systems merely make different trade-offs along a single continuum and among a set of related security primitives. We use a trace from a time-sharing UNIX server used by a medium-sized workgroup to quantify the costs associated with each of these secure storage systems. We show that encrypt-on-disk systems offer both increased security and improved performance over encrypt-on-wire in the traced environment.

1 Introduction

Much of the focus of recent storage security work has been on protecting communication between clients and servers in an untrusted, networked world [Gobioff98, Kent98, Mazieres99, Satran01]. In particular, the focus is on protecting data *integrity*: preventing unauthorized modification of commands or data, modification of requests in transit, and replaying of requests. Some of these systems further address the issue of *privacy*, or confidentiality, of data transfer: preventing the leaking of data in transit by snooping on the network.

The most comprehensive treatment of this topic is Network-Attached Secure Disks (NASD) [Gobioff99a], which uses capabilities provided to users by a file manager separate from the storage servers. A barrier to wide acceptance of the NASD scheme is the performance cost of the encryption and integrity checking needed at both clients and servers. In order to reduce this cost, NASD proposes a scheme using *pre-computed checksums* with secure hashes [Gobioff99] that pre-calculates and stores checksums for long-lived data. NASD does not provide a comparable scheme to optimize privacy since this would require *pre-computed encryption*. If data were stored on the server in encrypted form, then it would not be necessary to encrypt it for each transfer on the network. The difficulty with such a scheme is that encryption in NASD is done using session keys generated for each client/server interaction, whereas pre-computation requires longer-lived keys.

From the client's point of view, these two schemes are identical – it receives encrypted data and must pay the cost of checksumming and 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. Similarly, if written data is encrypted before it leaves the client and is stored encrypted, the server eliminates any decryption work.

Storing data in encrypted form 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 different purpose – 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 is no need to encrypt data again when it is sent on the network. Encryption is done by the original creator of the file, and updated by subsequent writers, but the server performs no encryption or decryption. Secure checksums are still needed to ensure the integrity of the communication, but privacy is ensured without repeated per-byte encryption[1]. In order to use the data, users must still decrypt it, but using a long-term key that must now be obtained *a priori*.

To support sharing in a system that encrypts data on disk, the problem is simply one of key distribution – how users obtain these long-term keys. This can be done via a centralized server such as the NASD file manager or an NIS server. Alternatively, a distributed scheme where data owners provide keys to eventual users directly, as would have to be done for a system such as CFS, removes a cen-

[1] If desired, privacy of arguments still requires encryption of message headers, but not of bulk file data.

tral point for attack. A variant of such a key distribution scheme is proposed in SFS [Mazieres99, Fu00] and further expanded in the Cepheus file system [Fu99]. The SNAD system [Miller02] combines aspects of both CFS (on-disk encryption) and SFS (secure communication and authentication) into a single encrypt-on-disk system.

Even though many secure storage systems have been proposed and described individually, there is no systematic way to compare and contrast them. We remedy this situation by presenting an agnostic framework to describe the features of these systems and the level of security they offer. Any secure storage system must implement a core set of functions, although they may vary in the detailed design choices. These choices affect both the level of security that the system provides, and the performance the system achieves. A similar study has been done to establish a framework for evaluating digital certificate revocation mechanisms [Iliadis00].

In addition to security and performance, there is a third factor to consider when building any secure system: the level of inconvenience users are willing to tolerate. If users must type in a separate password for every document they open, or individually choose access rights for every file they create, they will soon begin to circumvent the best intentions of the system designers [Whitten99]. Precise metrics to gauge the impact of this effect are not yet established, so we will treat this issue only indirectly.

Given our framework, we show how to quantitatively compare the performance of previously proposed systems, the overhead on users, and the security guarantees that the systems offer. We do this using a trace from a non-secure UNIX file system to estimate the work required for the various secure schemes. This evaluation is independent of the actual system implementations, and provides a general way of evaluating security and estimating cost. Finally, our analysis shows that encrypt-on-disk systems are not only more secure but also provide better performance than encrypt-on-wire systems.

The rest of the paper is organized as follows. Section 2 defines our framework for storage system security, identifies a range of attacks, and suggests a core set of security primitives. Section 3 describes how system designs proposed elsewhere fit into the framework, and how the choices they make impact security or improve performance. Section 4 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 day-to-day usage. Finally we conclude in Section 5.

2 Framework of storage security

In this section, we abstract the commonalities among known secure storage systems into a general framework. The framework consists of five components: players, attacks, security primitives, granularity of protection, and user inconvenience. We elaborate each of these next.

2.1 Players

Here we define the players we use in the rest of the paper. This list covers all of the possible players that one has to consider for protecting stored data. Each player is listed with a set of legitimate actions it can perform. Any other action by that player is treated as an attack.

a) ***owners*** – create and destroy data (i.e., render data un-readable by all readers), delegate read and write permission to other players, and revoke another user's privilege to read or write owned data.

b) ***readers*** – read data once permission to read was delegated by owners.

c) ***writers*** – modify data once permission to write was delegated by owners.

d) ***wire*** – transfers data between other players.

e) ***storage servers*** – store and return data upon request. (For instance, these are file servers in NFS, disks in NASD, or disk arrays in iSCSI.)

f) ***group servers*** – authenticate other players and authorize access based on membership groups as defined by *owners*. (For instance, these are group servers in NASD or the NIS server in NFS.)

g) ***namespace servers*** – allow traversal of namespaces, such as provide support for lookup of directories and files in directories.

Finally, we define an *adversary* to be any entity who attempts to perform functions other than those that it is authorized to. Notice that this definition of adversary also includes legitimate players attempting to perform actions beyond what they are authorized to.

Though in the above definitions, functionalities differentiate players, actual systems might choose to aggregate multiple players into a single entity. For instance, NASD combines the functionality of the group server and namespace server into a single metadata server.

We intentionally omitted any key-escrow agent from the list of players because its main purpose is to reveal keys and identities when necessary but it does not add to the basic level of security of a storage system.

At this point, it is important to note that the framework presented here is not intended to allow evaluation of the end-to-end security of a particular system. This requires careful analysis of each system component and the particular combination of components. Any secure system is only as strong as its weakest link. Our framework is no replacement for such analysis, but simply seeks to allow a high-level comparison among different schemes, purposely leaving some secondary details unexamined.

2.2 Attacks

Broadly, there are two kinds of data that players handle:

- short lived data that is communicated, or agreed upon, in each session, and
- long lived data and metadata for persistent storage.

Existing systems for network security have mostly addressed the compromise of short-lived data and the protocols used to communicate them. In addition to securing data on the wire, storage systems must also secure long-lived data on the servers. These two requirements give rise to the following set of attacks. The attacks may be mounted on the data or the metadata, unless explicitly specified otherwise:

a) ***by the adversary on the wire*** – for instance, an attack mounted on the NASD protocol used to communicate files to the clients.

b) ***by the adversary on the servers*** – for instance, an adversary updating a file on a NFS file server.

c) ***by a revoked user on the servers*** – for instance where a revoked reader (no longer part of the system) can continue to read files in Cepheus.

d) ***by the adversary colluding with the storage server*** – for instance, one where a CFS encrypted directory is deleted by the UNIX file system.

e) ***by the adversary colluding with the group server*** – for instance an adversary gaining access to data after corrupting a NASD file manager.

f) ***by the adversary colluding with readers or writers*** – for instance, a reader passing a copy of a file to an adversary.

The last attack, involving collusion with other readers or writers is very difficult to prevent without substantial complexity and support from outside the system, and has been listed above for completeness. We will not consider it further in this paper.

Each of the above attacks can further be broken into three kinds based on the effect they have on the data:

a) ***leak attacks*** – are those where the adversary gains access to some data.

b) ***change attacks*** – are those where the adversary makes valid modifications to data (i.e., modifications that readers cannot detect as invalid).

c) ***destroy attacks*** – are those where the adversary makes invalid modifications to some stored data. An invalid modification is any change to data that is detectable as incorrect by the owner or readers.

Table 1 provides a summary of these attacks and where they occur in practice. The data summarizes a survey of CIOs and system managers showing the percentage of respondents reporting a particular attack. The table shows the primary types of attacks from our list above that each of these real-world attacks touches. The intent is to motivate the importance of all of the attacks listed above, including some that may not have been considered very crucial in past work (such as revocation).

2.3 Core security primitives

Secure storage systems as proposed in research and commercial systems implement a myriad of security features to enable players to securely perform their functions. Though the details of the schemes used differ, the core

	frequency	*cost*	*messages*		*data*				
attack	*% of companies reporting*	*estimated damage ($ millions)*	*leak*	*change*	*leak*	*change*	*destroy*	*revoked user*	*denial of service*
telecom eavesdropping	10 %	1	X	–	–	–	–	–	–
active wiretap	2 %	n/m	–	X	–	–	–	–	–
system penetration	40 %	19	X	X	X	X	X	–	–
laptop theft	64 %	9	–	–	X	–	X	–	–
theft of proprietary information	26 %	150	–	–	X	–	–	X	–
unauthorized access by insiders	49 %	6	–	–	X	X	–	X	–
sabotage	18 %	5	–	–	–	–	X	–	X
virus	94 %	45	–	–	–	–	X	–	–
denial of service	36 %	4	–	–	–	–	–	–	X

Table 1. *Frequency and cost of attacks.* The frequency of various attacks and their mapping into our framework. The % numbers are as reported in a survey of five hundred system managers taken in Spring 2001, with almost all categories showing significant increases over previous years [Power01]. The cost column gives the self-estimated damage to their businesses. Note that although over 75% of respondents claimed that they had experienced some monetary damage due to the attacks reported, only 35% were able to estimate the extent of the damage, which means the numbers shown are only low estimates. Industry estimates of the total damage to companies worldwide from all attacks run into the billions of dollars. The boxes marked "X" show the primary damage caused by a particular attack, although other damage is also possible in many cases. The intent is not to exhaustively enumerate the damage, but to motivate each of the attacks in the framework as an important threat and give a very rough idea of relative importance.

set of security primitives can be abstracted into six types: authentication, authorization, securing data on the wire, securing data on the disk, key distribution, and revocation. As we show in Section 3, not all systems necessarily provide support for all of these, and the choices made directly affect the performance of the system and security guarantees provided. In the rest of this section we elaborate on each of these primitives and the important choices in implementing them.

2.3.1 Authentication

The purpose of authentication is to establish the identity of a particular player in order to authorize their actions. Storage systems may implement authentication in one of two general ways:

a) ***distributed authentication*** – owners explicitly authenticate each player to authorize access to the data they own (as in CFS, or the use of server public keys in SFS).

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

In general, there are three mechanisms to achieve mutual authentication: a public key infrastructure (PKI), a centralized scheme (e.g., Kerberos [Steiner88]), or a password-based scheme. The former two are quite similar. Both need a trusted third party and differ in how often this party is consulted. The latter one requires some pre-exchanged shared secret, which can be difficult to maintain in a distributed environment.

The usual concern is about authentication of owners, readers, and writers to storage servers or group servers, but there may also be concern about authenticating *servers* to *users* to prevent improper service. Again, although this is an important consideration, we do not consider it a primary security requirement for this analysis

2.3.2 Authorization

The purpose of authorization is to allow the owner of some data to delegate (partial) access to the data to another player. The user is authenticated and the identity checked against a known set of permissions determined by data owners. Authorization can be done in one of two general ways:

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

b) ***owner-handled*** – 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).

2.3.3 Securing data on the wire

Protocols for ensuring reliable and secure passing of messages have been well studied. Several standard protocols have been proposed, including SSL to protect web traffic, SSH to protect remote terminals, and IPsec to protect Internet traffic more generally [Kent98]. A variant of such a system for storage is used in NASD [Gobioff98]; a similar scheme is used in the self-certifying file system [Mazieres00]; and IPsec has been proposed as the security mechanism for iSCSI [Satran01].

To ensure data integrity on the wire some scheme involving keyed checksums (MACs) will always be needed, irrespective of the design chosen. The MAC is used to tie the checksum to a particular player, and the checksum is used to tie the MAC to a particular set of data. A timestamped MAC also protects against replay or server impersonation (man-in-the-middle) attacks [Gobioff98].

With the increasing deployment of protocols such as SSL and IPsec, hardware solutions are becoming available that offload the heavyweight cryptographic operations from client or server processors. Such hardware may support an entire protocol in its end-to-end form, or simply provide accelerated primitives that can be used in different ways by various systems. Once concerns over raw encryption or checksum speed are removed, parameters such as number of key changes and requirements for key storage present further bottlenecks [Cravotta01].

2.3.4 Securing data on disk

The reasons one may want to encrypt data on the disk are that the server is inherently untrusted or the server might be compromised, such as a stolen disk or laptop. To guarantee that the data and metadata are not compromised, they must be stored encrypted on disk. To accomplish this encryption, two types of ciphers may be used:

a) ***symmetric cipher*** – a single private-key system, such as DES or AES [Schneier95, Nechvatal00], that is used to perform bulk data encryption and decryption (such as the privacy option in NASD).

b) ***asymmetric cipher*** – a system using a pair of keys, such as RSA [Schneier95], that is generally used for authentication and to bootstrap the shared keys to be used by the symmetric cipher (such as the authentication protocols of IPsec).

Since computing asymmetric ciphers is much slower than symmetric ciphers, these operations are used sparingly, either for key exchange, or to protect stored symmetric keys in a *lockbox* (such as those used in Cepheus).

2.3.5 Key distribution

In a secure storage system that relies on encryption to protect data, each piece of data has some associated keys – either symmetric of asymmetric, depending on the structure of the system – that are required to access it. These keys may be used in one of two ways to:

a) ***directly encrypt data*** – the keys are used by writers to encrypt and by readers to decrypt data directly at the edges of the system (as in CFS).

b) ***prove authorization*** – possession of the keys is used by readers and writers to prove that they have the requisite authorization (such as the capabilities in NASD).

Use of the keys to prove authorization requires the server be trusted to accurately perform the necessary checks. Direct encryption ensures that only readers or writers are able to access the data or create valid new data. However, it complicates revocation since readers and writers have been given the keys themselves, rather than simply delegated capabilities.

2.3.6 Key distribution

For either use of keys, any system with shared access to files then requires some mechanism to distribute keys among readers and writers. Current systems implement this key distribution in one of two ways:

a) ***using a group-server*** – a centralized group server maintains the keys to all files, and the access control lists. If a user is in a particular list, then the server provides the key to the corresponding file (as in NFS, AFS, and Cepheus).

b) ***owner-handled*** – file owners themselves provide readers and writers with keys that they can use to perform actions. This typically complicates key revocation if the readers and writers cache keys (as in variants of CFS).

2.3.7 Revocation

Traditionally revocation is discussed in the context of centralized services such as certificate authorities (CAs) where it removes the association between the physical identity of a player and a particular key. In the context of secure storage, this is extended so that a player's access privileges to a particular piece of data can be revoked. When a player is revoked (e.g., a user leaves a particular workgroup) the keys to which this player had access must be changed. In systems where data is stored encrypted, this will require data to be re-encrypted, which may be done as follows:

a) ***aggressive re-encryption*** – immediately after a revocation, re-write data with a new key. Copies of data distributed under the old key in the past remain readable.

b) ***lazy re-encryption*** – delay re-encryption of the file to the next time it is updated [Fu99] or read. This saves encryption work for rarely-accessed files, but leaves data vulnerable longer.

c) ***periodic re-encryption*** – change keys and re-write data periodically to limit the window of vulnerability [Gobioff99a].

The distinction between aggressive and lazy re-encryption is a general consideration for secure storage. If a user had access to particular data at one time, they may anyway have copied it elsewhere, so protecting future changes becomes most important.

2.4 Granularity of protection

To provide secure storage, a system bears the additional overhead of the cryptographic operations discussed above, and the key management. To limit the key overhead, various systems implement different optimizations including aggregation of players into groups to simplify authorization, and trading off the security of short-lived keys against the ease of management of long-term keys.

2.4.1 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 player in a group, and then authorizing access based on group permissions. There are two ways to decide group membership, namely:

a) ***distributed group membership*** – owners explicitly determine who is authorized to share data and distribute the necessary keys (as in CFS).

b) ***centralized group membership*** – owners delegate authorization to a group server that distributes keys (as in NFS with NIS, NASD, and Cepheus).

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

2.4.2 Granularity of keys

The keys used to encrypt and decrypt a particular set of data may be short-term or long-term. Short-term keys reduce the vulnerability window by decreasing the amount of data encrypted with the same key, whereas long-term keys are easier to manage since there are fewer of them, and they are exchanged less often.

a) ***short-term keys*** – typically last for the duration of one player and one session (as in NASD, and iSCSI with IPsec).

b) ***long-lived keys*** – typically last across sessions and might be the same across multiple players (as in CFS and SFS).

When using long-term keys, the granularity of data associated with a single key greatly impacts the number of keys required; the choices include a key per-file, per-directory, per-user-group, or per-file-group.

Additional concerns arise when considering very long-lived keys, such as digital signatures on documents that must last for many years [Maniatis02], or for backup tapes [Boneh96].

2.5 User inconvenience

In addition to security and performance, it is critical to consider the level of inconvenience users are willing to tolerate before they become sloppy and circumvent the intent of the system:

a) ***convenient*** – single login password and tokens derived from this.

b) ***inconvenient*** – compartmentalized access, multiple passwords for different services, passwords are re-entered frequently and changed regularly.

c) ***very inconvenient*** – resources are protected at a very low level (e.g., password per document opened or per application invocation).

Forcing users to remember long lists of passwords often leads to poor password choices, or sloppy password practices (e.g., post-it notes) [Adams99]. The problem is exacerbated when users handle keys explicitly and make encryption choices on their own [Whitten99].

Some of the password issues may be addressed by widespread use of smart cards. The difficulty is that this removes the main aspect of active user involvement in maintaining security. Users must be aware of security in some way, otherwise they will become complacent and assume the system is infallible. The parameters of these trade-offs are not yet well understood, but overall security of data may well hinge on such usability issues [Whitten99].

3 Secure storage systems

In this section we cast previously proposed designs for secure storage onto the framework described in Section 2. Where appropriate, we highlight the trust assumptions made by each design, and mention specific extensions proposed. Our intent is to evaluate each system against a common set of criteria. For this reason, we concentrate on those aspects that address the primary functions of a secure storage system. This does not mean that additional functions or characteristics of individual designs are less important. The overall security of a system must always be evaluated holistically: a system is only as secure as its weakest link.

In this same vein, we also assume that issues of operating system trust are dealt with separately. For all the discussion in this paper, readers, writers and owners should be thought of as the smallest possible *trusted core* surrounding a user [Dalton01]. If necessary, this may even be a smart card or other protected device that handles all encryption and key storage.

The comparison in Table 2 summarizes the characteristics of each of the systems presented in this section, and which attacks each system addresses.

3.1 CFS and similar systems

The first widely-known discussion of security for storage systems is the Cryptographic File System (CFS) [Blaze93]. In CFS a directory to be protected is encrypted using a secret key. The underlying data is then 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 yet another file. Normal utilities such as backup function without alteration; they never have access to the cleartext data. The system is implemented as a user-level NFS loopback mount, and files are decrypted when accessed.

CFS was designed as a secure *local* file system, so it lacks features for sharing encrypted files among users. The only way to share a protected file is to directly hand out keys for protected directories to other users. However, CFS does protect against attacks where the bits on disk are compromised, such as when a computer is stolen. The key characteristics of CFS are:

players

- owners, readers and writers are indistinguishable.
- the host file system acts as the storage server as well as the group server, in authorizing file access.
- namespace traversal is handled by readers and writers themselves.

trust assumptions

- the storage server is untrusted and does not access the keys, protecting against leak and modify attacks involving collusion with the storage server.
- the storage server is trusted to prevent destruction of data.

security primitives

- owners handle authentication when distributing keys to encrypted directories and files.
- authorization for read is done by passing keys to readers and writers.
- the host file system verifies the authorization of writers to overwrite existing data, but the validity of these modifications is assured only by having the proper key.
- writers encrypt data using a symmetric cipher before storing it on disk.
- there is no provision to protect data while on the wire, as CFS is essentially a local system.

- since CFS is designed for the local file system, distribution of keys is done directly by the owners.
- revocation requires immediate re-encryption of data, since a revoked user can collude with the storage server to attack the data.

granularity

- the local file system aggregates users into groups to authorize access, but there is no explicit decision on aggregating the keys used to encrypt data.
- long-lived keys are used on a per-directory basis.

CryptFS [Zadok98] extends CFS to be more efficient by building it as a stackable file system rather than a user level server. It attempts to make the system more resilient to attacks due to corruption of individual users by using session IDs and user IDs to index into the key table, rather than using only usernames. TCFS [Cattaneo97, Cattaneo01] uses a lockbox to store a single key (rather than per-directory keys), and encrypts only file data and file names; directory structures and other metadata are left un-encrypted. Beyond the implementation differences and varying key granularity, CryptFS, TCFS, and CFS are identical with respect to our framework. All of these systems are described for use on a local file system. They could also be used as mounts over a remote file system, with protection of the communication to the remote server. We consider only the simple, local case here.

A later generation CFS [Blaze94] includes a key escrow system. This is necessary to recover keys when they cannot be obtained from the owner, for instance, after an owner has left the organization. Truffles [Reiher93] uses an alternative method of handling this problem by splitting keys such that any *n* members of a group can collude to regenerate the key of a missing owner.

All of the above systems assume untrusted servers; keys are known only to the owners, readers and writers, and not trusted to the system itself. The key escrow system in CFS depends on trusting the key database, but not trusting the servers. Truffles distributes this trust so that a group of owners are trusted instead of a single database.

There are several systems that encrypt data on entire devices and transparently decrypt the data when it is accessed. These include Secure Drive [Swank95], Secure FileSystem [Gutmann96] and PGPdisk [NA98]. These systems are similar to CFS except that they themselves do not perform any authentication or authorization; they rely on the operating system for these primitives.

3.2 SFS

Most secure storage systems assume the servers to be part of the trusted infrastructure and concentrate on guaranteeing that the users accessing the servers are properly authenticated. The Secure File System (SFS) [Mazieres99] addresses the problem of *mutually* authen-

			adversary alone			*with storage server*			*by revoked user*		*other*	
system	*S#*	*message attacks*	*leak*	*change*	*destroy*	*leak*	*change*	*destroy*	*leak*	*change*	*subvert group server*	*denial of service*
CFS & similar	3.1	–	yes	yes	no	yes	yes	no	–	–	–	no
SFS	3.2	yes	yes	yes	yes	no	no	no	no	no	no	no
SFS-RO	3.3	yes	yes	yes	no	yes	yes	no	no	–	yes[2]	no
Cepheus	3.4	yes	yes	yes	yes	yes	yes	no	yes[1]	yes	no	no
SNAD	3.4	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no
NASD	3.5	yes	yes	yes	yes	no	no	no	yes	yes	no	no
iSCSI w/ IPsec	3.6	yes	yes	no	no	no	no	no	yes	yes	–	no
LUN security	3.7	no	no	no	no	no	no	no	no	no	–	no
AFS	3.8	yes	yes	yes	yes	no	no	no	yes	yes	yes	no
NFSv4	3.9	yes	yes	yes	yes	no	no	no	yes	yes	no	no
Windows EFS	3.10	–	yes	yes	no	yes	yes	no	–	–	no	no
PASIS/S4	3.11	–	–	–	yes	yes	yes	yes	–	–	–	no[4]
OceanStore	3.11	–	yes	yes	yes	yes	no[3]	yes	yes	yes	–	no[4]

Table 2. *Summary of security guarantees provided by different systems.* A "yes" means that the system prevents that particular attack; for instance, Cepheus prevents attacks that leak data by stealing the storage server because it encrypts on the media. A "no" means that the system fails to handle that particular attack. A dash means the attack is not applicable to that system. (1) Cepheus uses lazy revocation, which re-encrypts data only on the next update; this allows data to leak until is has been updated, making this a qualified "yes". (2) Subverting the group server does not open any additional vulnerabilities that are not already present from the adversary acting alone. (3) Since only a single replica is used by each reader, a reader colluding with a single storage server could cause another reader to see invalid modifications. (4) Although a request to a busy replica could be re-directed to other replicas, a combined attack on all the replicas could still be mounted.

ticating the servers and users. Authentication of the server is necessary to prevent an adversary spoofing the server, for instance, when the servers are part of a public infrastructure. One important tenet of SFS is that it is independent of the key distribution and authentication mechanisms. The characteristics of SFS are:

players

- owners, readers and writers are differentiated.
- the storage server also functions as the group server in authenticating users.

trust assumptions

- the storage server is trusted with the data and is vulnerable to leak or modify attacks by an adversary colluding with the server.

security primitives

- servers and users perform mutual authentication. Servers are authenticated using *self-certifying* pathnames to files. Self-certifying pathnames are similar to mount points in traditional NFS, except that they have the public key of the server embedded in them.
- the group server uses NFS style user authorization.
- a session key is used to protect all communication between the server and users.
- a distributed mechanism is used to obtain server keys (through self-certifying pathnames).
- revocation of servers requires readers to check a centralized *revocation list* of revoked servers.

granularity

- traditional UNIX style aggregation of users into groups helps simplify authorization.
- uses a session key to protect all communication

3.3 SFS-RO

SFS was extended in SFS-RO [Fu00] to support storage and retrieval of encrypted read-only data. This provides a solution to securely distribute widely-accessed data (such as application binary kits) over the Internet using individually insecure mirrors as storage servers. SFS-RO has the following characteristics:

players

- same as SFS except that there are no writers – only owners can modify the data that they have created.

trust assumptions

- the storage server is not trusted with the data and hence not vulnerable to leak or modify by the adversary in collusion with the server.

security primitives

- same as in SFS, except that data is stored encrypted on the disk. Data is signed and encrypted by the owners when it is stored. Readers can verify the integrity of data by verifying the signature.

granularity

- since the data is already encrypted on disk, there is no need to encrypt it again before transmission.

3.4 Cepheus and SNAD

The Cepheus system [Fu99] builds on SFS to develop a general purpose file system, while Secure Network-Attached Disks (SNAD) [Miller02] combines the functions of CFS and SFS. In particular, both systems keep files encrypted on disk, and include the ability to share and update the encrypted data. They differ only in a few areas, and have the following characteristics:

players

- owners, readers and writers are differentiated via specific authorization schemes for writes.
- Cepheus uses separate storage servers and a group server that distributes lockboxes. SNAD relies on public/private key pairs for groups and must use a group server to distribute these, but stores lockboxes directly on storage servers.

trust assumptions

- the storage server is not trusted with the data and hence not vulnerable to leak or modify attacks by an adversary in collusion with the server.
- the storage server holds file encryption keys in lockboxes that are encrypted. In Cepheus, only readers and writers hold the keys to lockboxes, preventing attacks in collusion with the group server. In SNAD, separate key pairs are used for groups, so the group server for these is vulnerable.
- revoked users can continue to decrypt files until the files are updated, at which point they are encrypted with a new key (*lazy revocation*). Revoked users cannot update or destroy data.

security primitives

- servers check user authentication and authorization via the lockboxes.
- both systems use keyed HMACs stored with the data to detect modify attacks.
- all data on the disk is encrypted by the users when it is written. Both systems use symmetric keys, making possible modify attacks where readers collude with storage servers to write data.
- a session key and checksums are used to protect all communication between the server and users.

- keys to lockboxes are distributed by the group servers, and individual user public and private keys are required via a public key infrastructure.
- Cepheus implements lazy revocation, where files are re-encrypted only when they are next updated; SNAD suggests use of a similar scheme.

optimizations

- though different keys encrypt different files in the same group, they are kept in lockboxes locked with the same group key, so users need only one key per group.
- long-term keys encrypt all files.
- both systems use block-level encryption (8 KB blocks for Cepheus, 4 KB for SNAD) to allow updates of the individual parts of larger files.

A recent extension to Cepheus and SFS also assumes untrusted servers, and further seeks to detect attacks by the server on the integrity of stored data [Mazieres01]. For instance, one can detect when the server provides different versions of the same file to different users.

3.5 NASD

Network-Attached Secure Disks (NASD) [Gobioff99a] proposes a distributed network of intelligent disks with a shared group server (that also handles metadata for directory traversals). Access for data objects on the disks is authorized by the group server who hands a capability to the user. The disk and group server share a key, and presented with the appropriate capability, the disk services the request. Data is stored in the clear on the disks, but all communication is encrypted. NASD has the following characteristics:

players

- owners, readers, and writers are differentiated.
- the group server and namespace server is integrated into a single metadata server (the file manager), which is clearly distinct from the storage servers.

trust assumptions

- all messages on the wire are encrypted.
- since data is stored in the clear on the storage servers, NASD is vulnerable to attacks in collusion with the storage server.
- since all authentication and authorization data is present in the metadata server, NASD is vulnerable to attacks in collusion with the metadata server.

security primitives

- the metadata server authenticates and authorizes clients by handing them capabilities, which are later verified by the storage server.
- data is encrypted on the wire, and integrity is guaranteed using a MAC on checksums.
- the centralized metadata server makes revocation fast.

trust assumptions

- owners delegate capability distribution to metadata servers. The storage and metadata server are assumed to be trusted; all data is stored in the clear.

granularity

- checksums and keyed MACs ensure the integrity of requests and data transfer between clients and servers.
- introduces a scheme of pre-computed checksums for stored data to reduce the computation of generating checksums on each individual request.

NASD for the first time suggests that individual disk drives directly participate in security protocols. This requires at a minimum strong checksums and keyed MACs for integrity, and optionally encryption and decryption for privacy.

3.6 iSCSI

iSCSI [Satran01] is a draft IETF standard to connect hosts to SCSI devices using TCP as the transport. Since devices may be used across the Internet, security is a major concern. There is a draft proposal [Klein00] to implement a security protocol within iSCSI to authenticate hosts and protect the integrity of commands on the wire. The main characteristics of this proposal are:

players

- there is no notion of individual users; readers, writers and owners are all the same as the host on which they operate. The protocol leaves the issue of authenticating and authorizing individual users to the host.
- there is no group or namespace server, only a storage server.

trust assumptions

- although the storage servers and hosts are mutually authenticated, data is not protected from the server; making it vulnerable to attacks involving collusion with the server.

security primitives

- servers and hosts authenticate using a public and private key mechanism.
- the server does not explicitly differentiate between reads and writes.
- data and commands are encrypted while on the wire using IPsec [Kent98].

- the key for authentication is distributed by an external mechanism.
- revocation is achieved by changing the access control list.

granularity

- session keys are negotiated on a per-login basis.

3.7 LUN security

Disk arrays aggregate the individual disks in the array into logical units (LUNs), which are then accessed by host systems through a host bus adapter (HBA). LUN security proposes to control the access of particular LUNs from different HBAs. This is facilitated by unique IDs on the HBAs and world wide unique numbers which identify them. The host operating system and device driver are trusted not to forge or spoof IDs.

LUN security can be implemented either at the host [HP01a], in the network switch [Brocade01], or at the storage controller [HP01]. The following is true in general of these solutions:

players

- there is no notion of individual users. One can designate read-only permission to some hosts.
- there is no group or namespace server.

trust assumptions

- all players are trusted to identify themselves correctly. The network and servers are also trusted.

security primitives

- typically, players are identified by their world wide number, and this is used for authentication.
- authorization can be performed by maintaining an access control list as follows:
  - at the hosts, by setting up the set of storage controllers that the host may contact;
  - on the wire, by controlling the port mapping at the network switches
  - at the storage server, by setting up the list of HBAs allowed to access each LUN.
- no encryption is performed on the wire or on disk.
- revocation is achieved by changing the access control list.

3.8 AFS

AFS [Howard88] is one of the first distributed file systems that specifically addressed security issues. AFS assumes untrusted users, and uses Kerberos to authenticate users to servers. At the beginning of a session, users obtain *tokens* from a Kerberos server, which authorizes them to access the storage servers. AFS servers verify the tokens and then do appropriate authorization based on group information maintained by a group server. A secure version of RPC is used to protect communication, though some questions have been raised regarding this [Gobioff99a]. The key characteristics of AFS are:

players

- AFS servers act both as storage servers and group servers; authentication is performed by a separate Kerberos server.
- readers, writers, and owners are differentiated based on access control lists at the storage server.

trust assumptions

- apart from the users and the network, all other players are assumed to be trusted. AFS is vulnerable to leak, modify, and destroy attacks in collusion with any of the servers.
- if a user’s group information is changed (or revoked) the user continues to have access to files in that group until the user’s token expires.

security primitives

- Kerberos authentication is used.
- the servers maintain per-directory access control lists to authorize accesses. The underlying UNIX file permissions are also applied locally.
- AFS does not encrypt on the disk, but RPC messages are secured.
- revocation is done by either changing the access control list or making appropriate changes in the Kerberos server.

security primitives

- though the authentication is centralized, authorization is distributed to the storage servers.
- as in UNIX, users groups are used to simplify authorization rules.

convenience

- single password login via Kerberos, tokens cached for 24 hours by default, often set shorter (e.g., 1 hour) for administrative accounts, or longer (e.g., 30 days) for long-running applications.

3.9 NFS

There have been a number of proposals to build a secure networked file system by providing a security layer on top of NFS. These include proposals to secure the RPC [Taylor86] and tunnelling NFS through SSH or SSL [Gerraty99] to protect data on the wire. The security assumptions and implications of these systems closely match those of AFS and NASD.

The recent NFSv4 specification [Shepler00] explicitly addresses the problem of securing the RPC mechanism. Currently it proposes at least three security mechanisms: one using Kerberos and two using a public key infrastructure. All these essentially set up a secure communication channel and enable mutual authentication. Interestingly, one of these mechanisms, low infrastructure public key mechanism - exploits the fact that the client authentication can proceed after establishing a secure channel, to reduce the PKI overhead. In this scheme only the server needs to have a public/private pair which authenticates the server and sets up a secure channel. NFSv4 also greatly expands the use of ACLs for access control, very similar to AFS ACLs.

3.10 Windows EFS

The Encrypting File System (EFS) for Windows is integrated into NTFS and supports securing data similar to CFS [Microsoft99]. To facilitate file sharing, EFS uses lockboxes to hold the key of the encrypted file. This lockbox contains the file encryption key protected by a public/private key. EFS supports key escrow by including a key recovery agent among the users allowed to access any file. EFS encrypts and decrypts data just prior to the disk, so some external network security solution is required to secure the data on the wire to a remote server. The characteristics of Windows EFS are as follows:

players

- owners, readers, and writers are differentiated.
- the operating system functions as the group, storage, and namespace server.

security primitives

- Windows primitives are used for authenticating and authorizing writes.
- data is stored encrypted on the disk.
- data is sent in the clear on the wire.
- a user's private key is used to get the file encryption key. Some external mechanism must exist to distribute users' public keys.
- revocation requires re-encrypting files with a new encryption key and re-encrypting the lockbox.
- revocation is achieved by changing the access control list

trust assumptions

- EFS is vulnerable to attacks on the wire if used without an external secure network solution.
- EFS secures against leak and modify attacks mounted in collusion with the server.
- if the private key of the key recovery agent is compromised, all files in the system are protected only by the server's authentication and authorization primitives.

optimizations

- user groups are used by the native Windows file access control lists.
- files are encrypted using a long-term key.

3.11 Survivable storage

Addressing destroy attacks in collusion with storage servers requires survivable storage, i.e., some mechanism to recover from the total loss of a storage server by keeping multiple copies of the data. Several projects currently underway attempt to address security and long-term protection on a much wider scale (in space and in time) than any existing system. PASIS considers storage where data integrity is maintained in the face of the destruction or compromise of some number of replicas [Wylie00] and OceanStore considers a world-wide set of encrypted replicas [Kubiatowicz00]. Another mechanism for protecting data from unauthorized modifications is to use versioning on the storage servers so that data can be reverted to a state before an intrusion, as proposed by S4 [Strunk00]. The most powerful system to protect against all types of destroy attacks might well use a combination of these two schemes, as Carnegie Mellon has proposed by using S4 as a file system on top of PASIS storage.

4 Evaluation

This section explores the costs of implementing the various design choices discussed above, and the impact of these choices on security. The purpose of presenting this data is to compare the relative costs of the systems discussed in Section 3 using a trace from a real system. This allows us to evaluate expensive operations such as full-bandwidth encryption, key distribution, and key generation in practice.

The basis for our evaluation is a 10-day trace of all file system accesses done by a medium-sized workgroup using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage

	12-hour	*10-day*
hours	12	240
requests	11.5 million	97.4 million
data moved	23 GB	129 GB
active users	23	32
user accounts	207	207
active files	111,000	969,000
total files	4.0 million	4.0 million
file systems	24	24

Table 3. *Overview of file system trace used for evaluation.* The 10-day trace covers a period in late 2000 from a Thursday to the following Saturday. The 12-hour subset covers 8am to 8pm on the first trace day.

space. The trace was collected by instrumenting the kernel to log all file system calls at the syscall interface. Since this is above the file buffer cache, the numbers shown will be pessimistic to any system that attempts to optimize server messages or key usage based on repeated access. Table 3 provides an overview of the trace.

Implementing each of these systems in the same environment, with the same users, in order to perform a controlled experiment would be prohibitively expensive. We use an analysis of the trace to estimate how the system would behave and compare the relative operation costs. This requires us to make some inferences about the design of the various systems that are not always specified – we highlight these assumptions when they might affect the comparison.

4.1 Security primitives

Table 4 shows the total number of cryptographic operations required for particular security primitives, depending on the granularity at which they are implemented. This clearly illustrates the difference between the on-the-wire and on-the-disk encryption systems. In NASD, the server bears the cost of both the checksums and the encryption (assuming the privacy security level). This cost is reduced somewhat by the pre-computed checksums, but the encryption cost remains high. Since a session key is computed for each client/server interaction, the same file sent to multiple clients must be encrypted each time. In CFS, on the other hand, data is encrypted by the clients before ever being sent to the server. This provides the same level of privacy when data is on the wire, but requires only checksums and signatures at the server, as shown for Cepheus.

4.2 Granularity of protection

The primary comparison among the encrypt-on-disk systems is the level of protection and complexity of key management, and how keys are aggregated to objects.

Table 5 gives counts for the total number of keys used in each of the three high-level classes of designs – using per-session keys, per-file keys,or per-group keys. The table shows the number of keys on a per-user basis for several representative users and system userids during the 12-hour trace period. The representative usernames listed include the busiest users in terms of key use and key distribution, as well as several system userids that own substantial numbers of files. The first three columns consider per-session keys as used in the encrypt-on-wire systems. The middle four columns consider per-file keys as a logical extreme. The last four columns consider a per-group key scheme such as that used in Cepheus. The table shows 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 UNIX permissions bits share a single key). We see that the number of keys

		total ops (10 days)		*peak load (1 minute)*		*NASD*	*CFS*	*SFS*	*Cepheus*
	operation	*messages (1,000s)*	*bytes (MB)*	*messages (req/s)*	*bandwidth (MB/s)*				
server - integrity	message signatures	97,400	n/a	6,600	n/a	X	–	X	X
	checksums	37,300	129,000	6,600	12.5	–	–	X	–
	pre-computed cksums	14,600	n/a	1	n/a	X	–	–	X
server - privacy	encryption (reads)	22,700	77,700	780	6.4	X	–	–	–
	decryption (write)	14,600	51,400	740	6.1	X	–	–	–
client - privacy	encrypt/decrypt	37,300	129,000	1,520	12.5	X	X	X	X
server - key exchanges	per request	97,400	–	6,600	–	–	–	–	–
	per open/close	7,700	–	433	–	X	–	X	X
	per logical volume	24	–	4 / min	–	–	X	–	–
group server - key distribution	per file	11,100	–	1,100	–	X	X	–	–
	per group	177	–	18	–	–	–	–	X
	per logical volume	24	–	1	–	–	–	X	–

Table 4. *Number of cryptographic operations at the server for each design.* The total number of cryptographic operations performed by the server over the course of the 10-day trace, and during the busiest 1 minute interval in the trace. Message signatures are calculated for every request, checksums only for READ and WRITE requests. Checksums and encryptions/decryptions have a per-byte cost, whereas key exchanges and distributions do not. When using pre-computed checksums, only WRITE operations incur server checksumming. The peak load in terms of messages is an interval filled almost entirely with STAT requests; the peak load in terms of bytes has a much smaller number of READ/WRITE requests. The main cost difference can be seen in the privacy rows. In the encrypt-on-wire systems, both server and client work is required, whereas the encrypt-on-disk systems do not require the server work. The granularity chosen for keys has a large effect on the number of messages required for key setup and for key distribution by the group server, as shown in the last six rows. The values in the peak load column give the total streaming and per-message performance required from the server and client processors, or by any hardware engine that might offload the cryptography. The final four columns specify which systems bear which costs; an “X” means that the system uses the indicated cryptographic operation.

	session keys			*per-file keys*				*per-group keys*			
user	*per request*	*per open/close*	*per filesys/lv*	*total*	*owner*	*non-owner*	*newly created*	*total*	*owner*	*non-owner*	*newly created*
alice	85,100	6,130	6	203	6	197	85	29	1	28	0
bob	8,200	755	4	22	3	19	7	18	2	16	0
charlie	158,000	10,400	5	429	32	397	120	49	6	43	0
dick	46,500	2,360	3	475	62	413	3	24	8	16	0
erik	1,450,000	44,200	7	1,060	571	486	46	43	10	33	0
root	16,000,000	681,000	17	109,000	16,700	92,600	10,400	756	75	681	0
news	1,670,000	264,000	3	103,000	103,000	42	79,200	13	6	7	0
others	945,000	57,100	15	2,300	1,620	684	219	221	84	137	0

Table 5. *Key use by readers and writers.* 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, and iSCSI); on a per-file basis (as in CFS); and on a per-group basis (as in Cepheus). The total number of per-file or per-group keys by username is separated into the total keys used, the 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. The row for "others" contains the totals for the thirteen additional usernames active during the 12-hour trace. The rows for usernames "wilkes", "frank" and "bin" that appear in the following table are ommitted here since those users were not active during the 12-hour trace and the columns read 0 across the entire row.

	per-file		*per-group*	
user	*files owned*	*keys distributed*	*groups owned*	*keys distributed*
wilkes	54,500	7,810	28	18
alice	19,400	31	13	5
bob	216,000	6,210	17	11
charlie	4,020	148	12	8
dick	13,700	114	13	9
erik	133,000	1,650	14	8
frank	64,900	23,000	32	17
bin	191,000	14,800	33	21
root	240,000	644	129	29
news	1,570,000	554	15	5
others	1,430,000	40,400	2,260	601

Table 6. *Key distribution by owners.* Assuming a system that uses per-file keys, how many keys must a particular owner send to other users. The "owned" columns show the totals for all the files or groups that exist in the file system, and the "distributed" columns show the number of keys sent out during the 12-hour trace. The row for "others" contains the totals for the approximately 200 additional usernames on the system.

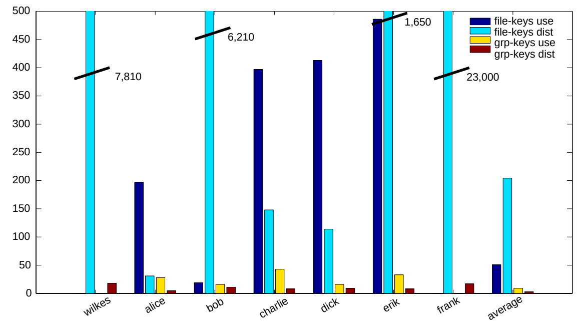

Figure 1. *Per-file vs. per-group keys.* The data of Table 5 and Table 6 presented graphically for several users. Using per-group keys dramatically reduces both the number of keys used by readers and writers and the number of keys that must be distributed by owners. Here "average" is the per-user mean of the "others" rows from the tables.

required for the per-group scheme is orders of magnitude lower than for the per-file scheme and several orders of magnitude less than most of the per-session schemes.

Considering the complexity for owners, as opposed to readers and writers, Table 6 looks at the number of keys that would have to be managed by data owners using per-file or per-group keys. The table shows the total number of keys needed by each owner. The "owned" column gives a count of all the files in the entire file system owned by the given user. The "distributed" numbers show the number of keys a given owner would have had to distribute during the time of the trace to readers and writers of the files for which they are responsible. We can see from these numbers that a system requiring direct user involvement for key distribution would be prohibitively cumbersome (imagine writing 7,500 keys from a possible list of 50,000 on scraps of paper in the course of several hours at your desk).

The two columns on the right are much more promising. They show the number of keys required if we move to a key-per-permission-group scheme. In this case, there is not a separate key for each file, but a key for each class of files, as described above. This produces a much more manageable list with roughly 30 keys per owner, with 10 or 15 of them distributed during a 12 hour period, something that could even be done manually (using scraps of paper) for maximum security. A graphical representation of the difference is shown in Figure 1 where the potential benefit of group keys is clear. An order of magnitude less keys are required for the per-group scheme.

	granularity	files	bytes	NASD	CFS	SFS	Cepheus
aggressive revocation	per-file	3,700	2 GB	–	X	X	–
	per-group	546,000	91 GB	–	–	–	–
lazy revocation	per-file	470	$^1/_2$ GB	–	–	–	–
	per-group	121,000	43 GB	–	–	–	X
encrypt-on-wire	per-session	969,000	129 GB	X	–	–	–

Table 7. *The cost of revocation for each design.* Note the explosion in the number of files that must be revoked when a per-group system is used. Aggressive revocation assumes that all affected files are re-encrypted immediately. Lazy revocation assumes that files are re-encrypted only the next time they are read or written, so the values show how much of the data had been re-encrypted after 10 days. This number would increase over time, eventually closing the window of vulnerable data and reaching the aggressive values. Note that even the aggressive, per-group scheme still performs less total encryption work than the encrypt-on-wire scheme which is constantly changing the keys. The final four columns specify which systems bear which costs, an "X" means that the system uses the indicated mechanism.

Note that the numbers in the table are skewed high since our analysis assumes users 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 (e.g., when we consider the entire 10-day trace, the total number of per-group keys distributed is, on average, roughly double the numbers shown for 12 hours). Another option would be per-directory keys as used in CFS. These numbers are not shown, but fall roughly between per-file and per-group keys.

4.3 Cost of revocation

The downside of using long-term keys for encryption is the additional cost on revocation. When a user leaves a group or organization and their access is to be removed, the stored data that is encrypted with any keys that the revoked user had access to must be re-encrypted to prevent future unauthorized access. Table 7 gives details on the cost of revocation when a user leaves a group. In a system that uses the same key for a group of files based on ownership or permissions, there is an additional revocation that results when a user changes permissions on a file (e.g., using *chmod* in UNIX), revocation for this reason is rare in our trace and not covered in the table.

We simulate revocation in our 10-day trace as follows. We choose a single user that will be revoked during this period[1] and track all the keys obtained by this user over the 10-day trace. For aggressive, per-file re-encryption, the number of files re-encrypted is simply all the files the revoked user accessed in the past 10 days. In a system with per-file keys, this is the total amount of data that must be re-encrypted. For a system with per-group keys, the cost includes the re-encryption of all the files in all the file groups to which the user had access. For lazy revocation systems, we assume that file data is re-encrypted as it is read from or written to the system. Data is re-encrypted and re-written whenever the file is accessed for read or write. These values are shown in the lazy revocation portion of the table.

[1] We believe that a frequency of one revocation in 10 days is reasonable. The turnover rate at Silicon Valley companies in the late 1990s averaged around 18% per year, which means that in a group of 200 people, a person would leave about every 10 days.

For the lazy revocation scenario in Table 7, the volume of data to be re-encrypted is nearly the same as the work done by an encrypt-on-wire scheme (the server encrypt/decrypt lines from Table 4). This gives further evidence for the duality between encrypt-on-the-wire and encrypt-on-disk schemes. In the encrypt-on-the-wire systems, data is encrypted and decrypted each time it crosses the network. In the encrypt-on-disk systems, data is already encrypted and requires no further work by the server. However, on revocation, the encrypt-on-disk system requires extensive re-encryption. With lazy revocation, this re-encryption occurs whenever the file is read or written, which makes the work done almost comparable to the encrypt-on-the-wire system. The only remaining difference is because encrypt-on-disk needs to perform the encryption only once (until the next revocation), whereas encrypt-on-wire repeats the encryption and decryption each time a file is transferred. The cost differential between the two systems will come down to the relative frequency of revocations, and the total amount of data a particular revocation affects.

5 Conclusions

This paper has developed a common framework of the core functions required for any secure storage system. We have reviewed all the previously proposed systems for storage security, and mapped them into this set of components and design choices. For integrity of network communication, any secure storage system must provide some variant of signed message checksums that strongly tie particular data to particular players. For privacy and confidentiality, we have shown that the two main classes of systems previously described are actually very similar: encrypt-on-wire (which solely protects the communication between servers and users) and encrypt-on-disk (which perform encryption and decryption only at user endpoints, with untrusted servers in between). The latter systems provide a form of *pre-computed encryption* for optimizing the encryption work done by the former systems. We have also shown that encrypt-on-disk systems with lazy re-encryption begin to have comparable encryption and decryption costs to encrypt-on-the-wire systems, even though these would seem to be completely different approaches at first glance.

We have quantified the costs of the various systems using a trace from a UNIX timesharing server and shown that the choice made about granularity of keys greatly affects both the complexity and encryption load – sometimes by orders of magnitude. We have quantified a number of design choices that affect security and performance: owner-based key distribution, precomputed encryption, the use of file groups, and lazy re-encryption. We have briefly mentioned survivable storage systems, but not analyzed their performance.

Our experience describing this framework has helped us focus our thinking on to how to build a comprehensive secure storage system that allows users to trade off their level of security and system performance in a concrete, sensible way. Our future work will follow these design choices and a sequel to this paper will report on a system that we are currently developing.

Acknowledgements

We thank all the users of SSP for allowing us to trace their behavior and report on it here. We thank Qian Wang and Jay Wylie for reading earlier drafts of this paper and asking questions we hadn't considered. We thank the anonymous reviewers, and above all our shepherd, Erez Zadok, for his tireless support, encouragement, and attention to detail; including the discussion in Section 2.5, which we would have forgotten about.

References

[Adams99] A. Adams and M. Sasse. Users are not the enemy: why users compromise security mechanisms and how to take remedial measures. *CACM* 42 (12), December 1999.

[Biham90] E. Biham and A. Shamir, Differential Cryptanalysis of DES-like Cryptosystems, *Advances in Cryptology (CRYPTO '90)*, August 1990.

[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.

[Boneh96] D. Boneh and R. Lipton. A Revocable Backup System. *USENIX Security Symposium*, July 1996.

[Brocade01] Brocade Communications Systems, Inc. Advancing Security in Storage Area Networks. *White Paper*, June 2001.

[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.

[Cattaneo01] G. Cattaneo, L. Catuogno, A. Del Sorbo and P. Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. *FREENIX 2001*, June 2001.

[Cravotta01] N. Cravotta. Accelerating high-speed encryption: one bottleneck after another. *EDN*, August 2001.

[Dalton01] C. Dalton and T.H. Choo. Trusted Linux: An Operating System Approach. *CACM* 44 (2), Feburary 2001.

[Fu99] K. Fu. Group sharing 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.

[Gerraty99] S. Gerraty. sNFS: secure NFS. *www.quick.com.au/Products/sNFS.html*, October 1999.

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

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

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

[Gutmann96] P. Gutmann. Secure FileSystem (SFS). *www.cs.auckland.ac.nz/~pgut001/sfs*, September 1996.

[Howard88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West. Scale and performance in a distributed file system. *ACM TOCS* 6 (1), February 1988.

[HP01] Hewlett-Packard Company. hp surestore secure manager xp. *www.hp.com/go/storage*, March 2001.

[HP01a] Hewlett-Packard Company. hp OpenView storage allocator. *www.openview.hp.com*, October 2001.

[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. Atkinso. Security Architecture for the Internet Protocol, *RFC 2401*, November 1998.

[Klein00] Y. Klein and E. Felstaine. Internet draft of iSCSI security protocol. *www.eng.tau.ac.il/~klein/ietf/ietf-klein-iscsi-security-00.txt*, July 2000

[Kubiatowicz00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *ASPLOS*, December 2000.

[Liadis00] J. Iliadis, D. Spinellis, D. Gritzalis, and B. Praneel. Evaluating certificate status information mechanisms. *CCS'00*, 2000.

[Maniatis02] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. *FAST*, January 2002.

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

[Mazieres01] D. Mazieres and D. Shasha. Don't trust your file server. *HotOS*, May 2001.

[Microsoft99] Microsoft Corporation. Encrypting File System for Windows 2000. *www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp*, July 1999.

[Miller02] E. Miller, D. Long, W. Freeman and B. Reed. Strong Security for Distributed File Systems. *FAST*, January 2002.

[NA98] Network Associates, Inc. PGPdisk (formerly CryptDisk). *www.pgpi.org/products/pgpdisk*, 1998.

[Nechvatal00] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti and E. Roback. Report on the Development of the Advanced Encryption Standard (AES). *National Institute of Standards and Technology*, October 2000.

[Power01] R. Power, 2001 CSI/FBI Computer Crime and Security Survey, *Computer Security Issues & Trends* VII (1), Computer Security Institute, Spring 2001.

[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, D. Smith, K. Meth, O. Biran, J. Hafner, C. Sapuntzakis, M. Bakke, M. Wakeley, L. Dalle Ore, P. Von Stamwitz, R. Haagens, M. Chadalapaka, E. Zeidner and Y. Klein. IPS Internet Draft - iSCSI. *www.ietf.org/ internet-drafts/draft-ietf-ips-iscsi-08.txt*, September 2001.

[Shepler00] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler and D. Noveck. NFS Version 4 Protocol. *RFC 3010*, December 2000.

[Steiner88] J.G. Steiner, C. Neuman and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. *Winter USENIX*, 1988.

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

[Swank93] E. Swank. SecureDrive. *www.stack.nl/~galactus/ remailers/securedrive.html*, May 1993.

[Taylor86] B. Taylor and D. Goldberg. Secure RPC: Secure networking in the Sun environment. *Summer USENIX*, June 1986.

[Whitten99] A. Whitten and J.D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. *USENIX Security Symposium*, August 1999.

[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.