Securing Distributed Storage: Challenges, Techniques, and Systems

Vishal Kher
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
vkher@cs.umn.edu

Yongdae Kim
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
kyd@cs.umn.edu

ABSTRACT

The rapid increase of sensitive data and the growing number of government regulations that require longterm data retention and protection have forced enterprises to pay serious attention to storage security. In this paper, we discuss important security issues related to storage and present a comprehensive survey of the security services provided by the existing storage systems. We cover a broad range of the storage security literature, present a critical review of the existing solutions, compare them, and highlight potential research issues.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection - Access controls, Authentication, Information flow controls; D.4.3 [**Operating Systems**]: File Systems Management - Access methods, Distributed file systems; D.4.2 [**Operating Systems**]: Storage Management - Secondary storage; E.3 [**Data**]: Public key cryptosystems; H.3.0 [**Information Storage and Retrieval**]: General

General Terms

Security, Management

Keywords

Authorization, Privacy, Confidentiality, Integrity, Intrusion detection

1. INTRODUCTION

Storage networks have become critical components of corporate computing environments and are evolving into complex, networked and distributed storage models. With the ever increasing growth in the number of organizations resorting to electronic data and online access, as well as in the number of data intensive applications, the sheer volume of data generated by these organizations is humongous. Further, this data has to be shared, replicated, and kept online in order to satisfy various performance, availability, and recovery requirements. As a result storage systems are becoming more vulnerable to security breaches, which can result in damaging losses.

There is a rapid increase in sensitive data, such as healthcare records, customer records or financial data. Protecting such data while in transit as well as while at rest is crucial. During its life-cycle, the data travels from various users, through various networks and storage systems, and ends up in online or offline data archives. Therefore, there exists a lot of potential attack points. Hence, data needs to reliably stored and protected at every stage of it's life-cycle. Recent regulatory requirements, such as Sarbanes-Oxley [6], HIPAA [3], DOD 5015.2-STD [5], and the European Data Privacy Directive [2], require data to be retained and secured for a longer period of time. At the same time, recent trends in data outsourcing have raised many new privacy issues making security a crucial requirement for storage systems.

In order to bolster the existing and future storage systems, one crucial step is to look back and understand what has been done in the past two decades in terms of storage security. From this we can learn the evolution of storage security, the current status, and the loop holes and missing points that one needs to address. Satyanarayan had presented a survey of distributed file systems [72] in the year 1990; however, there have been a lot of advances since then. Riedel et al. [69] presented a nice framework for evaluating security of storage systems. However, a comprehensive survey, critical review and comparison of current research is still missing. This paper presents a comprehensive survey of the security services provided by the existing storage systems. In particular, we answer the following questions: 1) what are the security services that should be provided by a storage system? 2) what kind of security services have been provided by the current storage systems and how? 3) what problems do the current solutions face? 4) what are the differences among existing solutions? 5) what new practical issues need to be resolved?

To answer the above questions, in section 1.1 we introduce various security services a secure storage system should consider. In order to depict the evolution of security in storage systems, distinguish existing systems using their inherent characteristics, and facilitate comprehension of this document we present a classification of the current storage systems in section 1.2. Sections 2–4 present case studies and critical review of the existing systems attempting to secure storage. We then compare these systems and raise a few practical questions in section 5. Finally, section 6 concludes this paper.

1.1 Security Services

Authentication and Authorization Data should be secured during its entire life-cycle. Authentication and authorization are the the most basic security services that any storage system should support. *Authentication* is defined as the process of corroborating the identity of an entity (called entity authentication or identification) or the source of a message (also called message authentication). The storage servers should verify the identity of the producers, consumers, and the administrators before granting them appropriate access (e.g., read or write) to the data. The act of granting appropriate privileges to the users is called *authorization*. Authentication can be mutual; that is, the producers and consumers of the data may want to authenticate the storage servers to establish a reciprocal trust relationship. Message authentication is performed by an entity to authenticate the origin of messages (or data) sent by another entity, for example, RPC messages (and data) transferred between a file server and a client. Authentication can be performed using various techniques such as passwords, digital signatures or message authentication codes (MAC) [50]. Authorization can be performed by maintaining access control list (ACL) on the storage server (e.g., UNIX ACL or Logical Unit Number masks) or by using capability certificates (or credentials) that list the access rights bestowed to the holder of the certificate.

Availability Most of the businesses require continuous data *availability*. System failures and denial of service attacks (DoS) are very difficult to prevent. A system that embeds strong cryptographic techniques, but does not ensure availability, backup, and recovery are of little use. Typically, systems are made fault tolerant by replicating data or entities that are considered as central point of failure. However, replication incurs a high cost of maintaining the consistency between replicas.

Confidentiality and Integrity As the data gets produced, transferred, and stored at one or more remote storage servers, it becomes vulnerable to unauthorized disclosures, unauthorized modifications, and replay attacks. An attacker can change or modify the data while traveling through the network or when the data is stored on disks or tapes. Further, a malicious server can replace current files with valid old versions [46]. Therefore, securing data while in transit as well as when it resides on physical media is crucial. *Confidentiality* of data from unauthorized users can be achieved by using encryption, while data *integrity* (which addresses the unauthorized alteration of data) can be achieved using digital signatures and message authentication codes. Replay attacks, where an adversary replays old sessions, can be prevented by ensuring freshness of data by making each instance of the data unique. This can be achieved by marking each session with timestamps or random numbers. Data can be secured while in transit by using protocols such as SSL [25] and IPsec [42]. In this case, the data is secured during transit, but the server decrypts the data before storing it on the disks.

As demonstrated by the recent incidences of data theft, confidentiality and integrity of the data and the meta-data must be ensured not only while the data is in transit, but also while the data is stored on the physical medium. Confidentiality and integrity of data at rest, as well as while in transit, can be achieved by performing cryptographic operations on the users' side. This is called *end-to-end security* where the writers encrypt (and sign) before sending the data to the storage servers and the readers decrypt (and verify the integrity of) the data on their machines. Encryption and decryption is not done on the server side. If the writers are required to sign their modifications, then the signatures also ensure *non-repudiation*, since the writers cannot deny their modifications. End-to-end security places minimal trust on the storage servers and the data is accessible only to the users with appropriate keys. Therefore, securing these keys is of paramount importance for the systems that provide end-to-end data security. Further, the keys have to be secured as long as the data is not deleted.

Key Sharing and Key Management In multi-user net-centric applications file sharing is quite common. If the files are encrypted, then in order to share files one has to also share keys. Efficient and scalable *management* of these keys is important, as revoking a user from a group (of users sharing files) or merging two groups can require re-encryption of shared files and re-distribution of new keys. Another important aspect of key management is key recovery, which is a technique for recovering lost keys. A *key recovery* system (see [20, 19] for details) is an encryption system with a backup decryption capability that allows authorized persons (users, officers of an organization, or government officials), under certain prescribed conditions, to decrypt ciphertext with the help of information supplied by one or more trusted parties who hold special data recovery keys. Further, special care has to be taken while storing, archiving, and deleting these keys.

Auditing and Intrusion Detection Storage systems must maintain *audit logs* of important activities. Audit logs are important for system recovery, intrusion detection, and computer forensics. Extensive research has been done in the field of intrusion detection [8]. Intrusion detection systems (IDS) use various logs (e.g., network logs and data access logs) and network streams (e.g., RPCs, network flows) for detecting and reporting attacks. Deploying IDS at various levels and correlating these events is important.

Usability, Manageability and Performance Finally, a storage system should have acceptable *usability*, *manageability*, and *performance* even while employing cryptographic primitives. Strong security with poor usability and/or performance will not make the system practical. As illustrated by [79, 39], secure systems and cryptographic softwares are used less than we would expect due to their lack of consideration for usability of their products. In other words, a secure storage system with complex management and poor usability will not be used (or will be used incorrectly) in practice.

1.2 Classification

In this paper, we survey various storage systems and the security services provided by these systems. We cover a broad range of storage systems, discuss the security services they provide (with reference to section 1.1), and discuss advantages and disadvantages of each. A subset of services discussed in section 1.1 are studied and compared. In particular, we do not consider availability, Byzantine failures, and performance (which will require real-time analysis) in this paper. The systems are classified based on their security services. The storage systems surveyed in this paper can be categorized as follows.

Networked File Systems In systems belonging to this category, the storage server authenticates each user and checks whether the user has appropriate privileges before granting any access to the data. Most of the systems belonging to this category also encrypt the network traffic. However, these systems do not provide end-to-end data security, that is they do not ensure integrity and confidentiality of the data stored by these servers. Instead, they assume that the file servers and the system administrators are trusted.

Traditionally, authentication was performed using passwords and access control was performed using simple UNIX-style access control (user and group identifiers). For example, earlier versions of the Network File System (NFS) [76] used a centralized server to authenticate clients and passed the user and group information to the clients. This information was then passed to the file server (during file requests), which then passed this information to the server's operating system who actually made the access control decisions. Thus, security was extremely primitive. File systems that paid more attention to security used better authentication mechanisms (such as Kerberos [57] or Public Key Infrastructure) and also encrypted the network traffic. For example the Andrew File System (AFS) [36, 70, 71] used a centralized Kerberos server for authentication and also encrypted the traffic between the client machines and the file servers. More recent file systems [49, 54] enable distributed authentication and cross-domain decentralized file sharing using Public Key Infrastructure (PKI).

Cryptographic File Systems The goal of these systems is to provide end-to-end security, where cryptographic operations are performed on the client side to keep data secure from the server as well as other unauthorized users (including administrators). These systems embed cryptographic operations (encryption/decryption and signing/verification) into the file system itself. The file server is minimally trusted and never sees the data in clear-text as it is not involved in the encryption or decryption process.[1]

These file systems can be further categorized into *shared* and *non-shared* file systems. Shared cryptographic file systems (e.g., [32], [40]) share files among a group of users. Therefore, these systems should complement the basic cryptographic services with more sophisticated key management techniques that efficiently handle key sharing and key revocation. On the other hand, cryptographic file systems that do not assume shared access to files do not include key sharing and key revocation mechanisms. In these systems (e.g., CFS [12]), in order to share files with other users the owner has to give away his/her private keys to other users, which gives them the same privileges as the owner. Some of the cryptographic shared file systems provide integrity of data and meta-data either by using MAC or by using digital signatures (which also provides non-repudiation).

[1]See [4] for recent standardization efforts by IEEE SISWG regarding cryptographic algorithms and methods for securing data at rest.

Storage-based IDS: A storage-based intrusion detection system is an intrusion detection system (IDS) embedded in a storage device or a file server. It analyzes data access patterns and data modification characteristics, looking for manifestation of an attack. The main advantage of a storage-based intrusion detection system running directly on the storage servers is that compromise of a host operating system does not result in compromise of the storage-based IDS; therefore, a storage-based IDS can still perform in the presence of host compromise. Further, the storage servers can perform inline detection by analyzing every request from the client.

In the following sections we examine the various storage systems. We briefly present the overview of these systems and focus on the security services provided by these systems. Some systems from each category are summarized in one section (titled Other Systems) due to page restrictions.

2. NETWORKED FILE SYSTEMS

2.1 Network-file System

Overview The Network File System (NFS) [76, 60, 74] is the most widely used network-attached file system. It enables heterogeneous clients to transparently share files stored on remote file servers without having to worry about the location of the files. An authorized client on a legitimate machine can mount the file system stored on the server. Heterogeneity and portability were the driving principles in the design of NFS. NFS has two basic components: the client program installed on the client machine and a server program (number of daemons) installed on the server machine. NFS client communicates with the NFS server using Remote Procedure Calls (RPC) [52], which allows one host to call functions on another host.

A system administrator can give access to the desired file systems by listing the *exported* directories in a file called `/etc/exports`. During the export operations, the administrator can specify a list of hosts that will be allowed to access the exported directories and the security flavors that a client can use to access the exported file systems. The client mounts the exported file systems by contacting the `mountd` daemon located on the server, which checks the export list and allows or disallows access to the client depending on the client's credentials. After mounting a file system, it is integrated into the client's directory tree and the client can access the mounted file system as if it is a local file system. NFS version 4 (NFSv4) [74] is the most recent version of NFS. It mandates many new changes to the previous versions. For example, all of the earlier versions were stateless and used different protocols to handle file mount and file locking operations. The NFSv4 protocol is a stateful protocol that integrates all the different protocols into one standard protocol. Further, unlike earlier versions, the NFSv4 protocols mandates the use of strong security mechanisms. An excellent overview of differences between NFSv4 and it's previous versions can be found in [60]. Below we give a brief overview of security services of the NFSv4 protocol.

Security Security was an afterthought in NFS. Most of the earlier NFS servers relied on the underlying operating system to perform access control and implemented weak authentication mechanisms. A variety of authentication flavors were defined for NFS version 2 [78] such as UNIX-style authentication (user ID and group ID), Diffie-Hellman based authentication, and Kerberos version 4 based authentication. However, even though the RPC mechanism allowed multiple authentication mechanisms, the UNIX-style authentication mechanism using UID and GID was most commonly implemented. This mechanism is insecure since an attacker can easily spoof (or replay) a user's credentials.

Therefore, NFSv4 mandates the use of the RPCSEC_GSS flavor [24]. RPCSEC_GSS provides authentication, confidentiality and integrity of the RPC requests and responses. Further, it can use security tokens from security mechanisms (such as Kerberos) that conform to the Generic Security Application Programming Interface (GSS-API) [47]. Conforming NFSv4 protocol implementations must implement Kerberos version 5 [48] and LIPKEY (A Lower Infrastructure Public Key Mechanism) [23]. Kerberos (briefly explained in section 2.2) can provide strong authentication within an Intranet. Thus, conforming NFSv4 protocol implementations will support strong authentication as well as network security (with some performance penalty). The LIPKEY protocol is intended to be used over the Internet. Every file server has a public key certificate and the associated private key. A user can setup a secure channel with the server by encrypting his login information with the server's public key and sending the information to the server. The user can also authenticate the server by verifying the server's certificate. During the authentication phase a symmetric key is established which will be used to secure the communications between the user and the file server. Thus, using LIPKEY a user will be able to access his files across the Internet.

Traditionally, NFS supported the POSIX [63] access control model. However, NFSv4 includes ACL support based on the Windows NT model and not the POSIX model, since the Windows NT model is richer than the POSIX model. An access control entry in Windows NT model are of four types with obvious meaning: ALLOW, DENY, AUDIT, and ALARM. Thus, NFSv4 provides strong security and flexible file sharing.

2.2 Andrew File System

Overview The Andrew File System [36, 70, 71, 58] was originally developed to provide a *scalable* campus-wide file system for home directories, which would run effectively using a limited bandwidth campus backbone network. The main services of AFS include scalability, caching, and simplicity of addressing.

AFS evolved further into a scalable distributed file system that enables co-operating hosts (clients and servers) to efficiently share file system resources across both local area and wide area networks. AFS supports completely autonomous cells, which represents an independent administered AFS site, for example, `cs.cmu.edu` is one AFS cell owned by the computer science department at CMU. A cell has its own protection domain, authentication servers, file servers, volume location servers, and system administrators. The local administrator can publish his own cell and make other cells visible to his local cell by listing all the necessary remote cells in the local cell's database server machines. AFS cells co-operate to support seamless cross-domain file sharing. A client can access a file stored at another cell by including the destination AFS cellname into the pathname as follows: `/afs/$cellname/filename`.

Security Authentication in AFS is performed using Kerberos [57] (a slight variation of Kerberos version 4). In order to access files stored in a cell, the client should have an account in that cell. The Kerberos authentication server maintains a database of users' passwords, which is encrypted using a key known only to the server (and the administrator). A user's password is never sent over the network, rather it is used to send encrypted authentication requests to the authentication server, which uses the user's password to verify and authenticate the user's request. After authenticating the user, the authentication server issues a ticket to the user, which is used by the user to authenticate herself with the file servers and vice-versa (mutual authentication). A session key is established during this mutual authentication, which is used to encrypt the traffic between user machine and the file server machine. AFS encrypts all the traffic between the user and the authentication server, and optionally encrypts all the traffic between the client and the file server. It is important to note that Kerberos is not effective against password guessing attacks.

Access control decisions are based on the protection domain, which is composed of *Users* and *Groups*. A group has a owner and set of users and other groups. If a user is a member of a group that is a member of another group, then the user inherits membership from both the groups. A user can create his own groups, the names of which are prefixed by the name of the creator of the groups. For example, a user *andrew* can create a group *andrew:friends* for his friends. The advantage of supporting group memberships and group inheritance is simplicity of management and revocation, and ease of administration of protection domains. Access control is achieved by using *Access Lists* associated with each directory, which can list users and groups, and their associated privileges. There are two types of access lists, namely, a list of *Positive Rights* and a list of *Negative Rights*. The negative access list always takes precedence over the positive access list. The access lists in AFS (read, write, lock, execute, delete, lookup, administer and insert) are quite sophisticated as compared to the standard UNIX-style ACL (read, write, execute) and it allows users to create their own groups and inherit group memberships.

The protection domain information has to be replicated to all file servers. Hence, if a user is removed from an access list by one server, the user can still send requests to other severs. In the case of a widely distributed network, discovering all groups that a user should be removed from and propagating this information to all the servers can incur significant overhead. Therefore, AFS first propagates negative access rights for that user on sensitive objects, which is faster. The user can then be removed gradually. Thus, there can be a small (potentially large due to network problems) window of time for which the protection domain on different file servers can remain inconsistent.

2.3 Self-certifying File System

Overview Most of the file systems (including NFS and AFS) have a notion of administrative realms and rely on central administration for configuration, management, and security purposes. However, the presence of a central entity hinders global file sharing and prevents easy addition of new file servers. AFS went one step ahead of NFS by enabling cross-cell file sharing. However, a user is required to have an account on the remote file server and has access to the files if and only if the user's cell is listed in the file server's local database. SFS aims to provide a secure global file systems with completely decentralized control. Anyone can set up an SFS server and any user can access any server from any client (if the user has right privileges).

SFS separates key management from the file system entirely by introducing *self-certifying pathnames*. A self-certifying pathname is a file name that effectively contains the appropriate remote server's public key. During file access, using the public key embedded in the pathname the SFS client can verify the authenticity of the SFS file server. SFS does not bind users to any particular key management technique and the users are free to choose any suitable technique to determine the authenticity of the public keys used by SFS (user and server public keys). The SFS architecture consists of three entities: the SFS file server, the SFS client, and the SFS authentication server that stores users' group membership and public key information.

Security Every SFS user and server has a public-private key pair. The SFS file system is accessible under `/sfs/Location/Hostid`. `Location` is the location of the file server, which can can be either a DNS hostname or an IP address. `HostID` is a cryptographic hash of the server's public key and the server's Location. The SFS client assumes that the pathname provided by the user is trusted and leaves the process of acquiring certified pathnames to the user. Trust in pathname implies trust in the server's public key. A `HostID` lets SFS clients ask a server for it's public key and verify the authenticity of the public key by verifying the hash of the server's public key with the `HostID`. SFS symbolic links maps human-readable names to self-certifying pathnames.

When a user requests for a file stored on a remote server, the SFS authentication agent located on the client's machine initiates an authentication protocol. As the first step of the authentication protocol, the client machine generates a public-private key pair. The client machine then acquires the public key of the SFS server and checks whether the hash of the key matches the `HostID` of the pathname. If yes, it establishes shared session keys, and achieves mutual authentication (with the server) using a protocol based on public key encryption. These session keys are used to establish a secure channel (encrypted and authenticated) between the client machine and the file server. After this, the authentication agent sends a signed request using the user's private key to the file server over the already established secure channel. The information sent to the server during the user authentication process is tied to the session; therefore, even if an attacker compromises the current session he cannot use the users' authentication information to setup new sessions. To ensure forward secrecy, SFS frequently changes the client machine's public key. After receiving the authentication request, the file server forwards the signed request to the *authentication server*, which verifies the signature and (if verified) passes the users' credentials (group memberships) to the file server. The file server caches these credentials and uses them to authenticate the user's future requests and to perform access control.

Management of public keys is not a part of SFS. This keeps the file system simple and scalable. Further, users have the flexibility of using any key management procedure of their choice, which is important for a global file system where different users may prefer different policies. SFS also supports some server key management schemes such as secure symbolic link and manual key distribution, certification authorities, and password authentication. Thus, SFS is flexible, scalable, ensures mutual authentication between the client machine and the file server, and secures all communication between them.

Decentralized Access Control Using SFS SFS is further extended to provide a decentralized access control mechanism [41]. This system allows users to grant access to specific users and groups belonging to local as well as remote administrative domains, without assuming any pre-existing administrative relationship. If a user Alice in University of Minnesota (UMN) wants to share files with some users from University of California Santa Barbara (UCSB), Alice first creates a group, adds local and UCSB users (or groups) to this group and places that group along with the ACL of the shared objects. Alice thus explicitly trusts the users or groups listed in the ACLs. The information regarding local and remote users and groups is used by the SFS authentication server local to UMN. In the case of remote users, the UMN authentication server fetches authentication related information (remote users public keys) from the remote authentication server of UCSB. The authentication servers are SFS servers with self-certifying pathnames. When a user accesses the shared files, the SFS authentication server hands over the credentials (group memberships) for that user to the file server using the the local information that was obtained from the remote SFS authentication server. Every authentication server periodically fetches the information regarding remote users from the remote authentication server by setting up a (long lived) secure connection with them using the self-certifying names.

This approach bestows the users with a privilege to grant access to any user trusted by them. Therefore, it is user-centric and trusts the insiders to grant access only to appropriate users. Further, since the authentication servers download the group membership lists periodically, updates to group memberships (addition or deletion) on remote sites may not reflect immediately in local authentication servers.

SFS-based Read-only File System SFS is extended to provide a fast and secure distributed read only file system [27]. The goal of this system is to securely distribute public, read-only data such as executable binaries or software distributions. The read-only file system is a SFS file system, which contains the server's public key as a part of the pathname (self-certifying). The SFS read only file system assures the client that the data retrieved from the server is authentic and is consistent with the current distribution, that is, the data is not older than the file system consistency period. It does not guarantee the confidentiality of the data stored on the server.

The read-only file system contains three programs, namely a database generator program, a client daemon, and a server daemon. The publisher of the read-only database runs the database generator program. This program hashes each data block of a file, inserts the hashed value of the data blocks in to the file inode, then hashes each inode and inserts the hashed value of the file inode into its parent directory and continues the hashing process until the root of the directory is hashed. In this manner, it creates a hashed tree. After hashing the root directory, database generator program signs the hashed value of the root directory (which is also the root of the hashed tree) using the private key of the publisher. The database is then replicated to various servers without requiring any further modifications. The client daemon handles the file system functionality on the client side. On receiving some data blocks from the server, the client can walk up the tree and verify the signature on the hash value of the root directory. In order to accomplish this, the client has to recursively request and check the validity of the parent blocks from the server.

2.4 Network Attached Storage Devices

Overview Most of the distributed file systems (including all of the systems discussed above) are implemented through one or more file servers that retrieve data from a storage network and deliver this data to the user. Every request goes through the file server, which can impose heavy load on the file server, thus affecting the scalability and performance of the system. Further, the devices attached to the storage network pay little attention to security and assume that security is handled by the file server. The NASD [7, 30, 31] project attempts to solve these problems by attaching the storage device directly to the network and removing the file server from the data path. Users can directly interact with the storage device, which gives improved performance to the end users. Since the storage devices are attached to the network and no longer protected by the file server, the storage devices are now responsible for authentication and access control operations. The NASD drives export an object-based interface as compared to the classical block-based interface.

NASD is a cryptographic capability system. A capability is a token that grants the bearer the access rights specified in that token. It has three main players: untrusted users, trusted filemanager that makes access control decisions, and trusted storage devices that authenticates users and implements the filemanager's access control decisions. It is assumed that the communication link between the filemanager and the users is secure and the filemanager has enough information to authenticate each user. The filemanager retains the responsibility for administering the name space and the access control policy of the file system, but the filemanager is bypassed in the common case operations, such as data transfer. The filemanager stores an access control list and shares a (set of) unique symmetric key with every storage device.

Security When a user wants to access a file, on the first access, the user sends a request for a particular object to the filemanager. On the receipt of the user's request, the filemanager authenticates the user and generates a capability key for that user. The capability key is derived by generating a MAC of the user's public credentials using the key shared between the filemanager and the storage device that stores the requested object. Apart from other entries, the user's public credentials contain the object-ID, the access rights of that user for the requested object and the expiry time of the capability key.

The filemanager transmits the capability key (on a secure channel) and the associated public credentials to the user. The user then sends the request (e.g., read) for the object (or collection of objects) along with the user's public credentials for that object, and a MAC of the request, which is generated using the capability key. After receiving the user's request the device can generate the capability key and verify the MAC of the request. The device can generate the user's capability key using the key shared with the file manager and the user's public credentials. If the MAC is verified, the device is assured that the capability key is generated by the filemanager. The device grants access to the user if the public credentials include the required privileges. Mutual authentication can be achieved by asking the device to compute a MAC of the responses using the capability key. Freshness of messages between the client and the storage device is ensured using timestamps. Therefore, NASD requires time synchronization between the client machines and the storage devices. In addition, the user can also choose to encrypt the data transmitted between the user and the storage device. Thus, along with authentication and access control, NASD can provide integrity, freshness, and privacy of data as well as user commands to the storage device.

The capability keys are typically short lived. If immediate revocation is required, the filemanager has to change the version number (which is also included in the public credential of the capability key) stored with the object or change the keys used to generate the capability keys. Changing the version number will revoke all the capability keys for that object generated using that version, whereas changing the shared key revokes all the capability keys generated using that shared key. Therefore, NASD does not allow fine grained revocation. For example, it is difficult to revoke all the capabilities (before they expire) generated for one particular user. Finally, since a user has to acquire a capability for every object, the filemanager has to remain online and can be a central point of failure.

The NASD project laid the foundation for network attached storage devices and gave birth to the Object-based storage technology (OSD). Work is currently underway to standardize the OSD command sets [59]. A number of extensions to NASD [66, 67, 9, 43] have also been proposed.

2.5 Other Systems

OceanStore [45, 68], a storage infrastructure developed at University of California, Berkeley, is designed to span the globe and provide scalable, continuous access to persistent information. Files (referred to as objects) are never deleted in OceanStore and an update to a file causes creation of a new file block. In order to ensure availability, scalability, and fault-tolerance, objects are replicated to multiple locations. Each object has an unique global identifier and location, and routing of replicas is done using the Tapestry [35] overlay network. Further, to keep the replicated copies consistent, OceanStore uses a combination of Byzantine update commitment amongst a subset of replicas of an object (called primary ring) and push-based update of other copies of that object (called secondary copies) using an overlay multicast network. While the initial OceanStore design includes end-to-end data security, Pond [68], the OceanStore prototype includes only authenticated Byzantine messages, but does not include end-to-end data security. In addition, the servers taking part in the Byzantine agreement (primary ring) employ threshold signature schemes [65] so that the users will have to verify only one signature and addition of new server to the ring will not require changing the ring's public key.

Distributed Credential FileSystem (DisCFS) [54] aims to allow a local user Alice (within a domain) to share files with an external user Bob that does not have an account on Alice's local file servers. The goal of DisCFS is to enable file sharing without administrative interference. In order to facilitate such file sharing, Alice's system administrator gives to Alice a *credential* for each file that Alice is allowed to access, which contains Alice's public key, file-id, Alice's access rights, administrators signature and some constraints. Alice's local file server only trusts the Administrator's signature. When Alice wants to access a file, Alice authenticates with the file server by sending a signed request and her credential. The server verifies Alice's signature, the Administrator's signature on Alice's credential, and if all the verifications are successful, grants Alice access to the file depending upon the access rights stored in the credential. Alice can delegate access rights to Bob by creating a similar credential for Bob and signing Bob's credential. When Bob sends a request for a file to Alice's local file server along with his and Alice's credential, the server will verify the chain of certificates by verifying Bob's signature, Alice's signature on Bob's credential, and the Administrator's signature on Alice's credential. Inodes are used as file IDs in the current implementation, which is subject to change as inodes are not suitable for unique global ID and can be reused. The DisCFS server has to verify a chain of certificates before granting access to a single file. This is likely to impose a lot of cryptographic overhead on the file server. Further, credentials are per-file; therefore, a user has to get multiple credentials from the administrator to access a file stored under a hierarchical directory (one per file and directory) structure.

3. CRYPTOGRAPHIC FILE SYSTEMS

3.1 Non-shared Cryptographic File Systems

3.1.1 Cryptographic File System for UNIX

Overview CFS [12, 13] designed by Matt Blaze is one of the first file system to perform file encryption. This system pushes the file encryption services into the file system. CFS is a virtual file system that runs at user level on the client machine. In particular, it mainly performs the file encryption and key management functions and leaves the rest of the functions to the underlying file systems. The main goal of CFS is to present the user with a secure file service that works in a seamless manner, without considering the encrypted files as special components of the system.

CFS is typically mounted in `/crypt`. Users can associate a key with a directory. All the files in this directory as well as their pathnames are encrypted using this key. The encrypted files can reside on a file server and the underlying file system could be any available file system including remote file servers, such as NFS. The user creates an encrypted directory using the `cmkdir` command. This command requires the user to enter an ASCII passphrase. CFS then creates a key using this passphrase. To use the directory created using the `cmkdir` command the user uses `cattach` command. This commands maps the encrypted directory to a virtual directory, which can be viewed under `/crypt`.

Security CFS uses a combination of OFB (output feedback mode) and ECB (electronic code book) [50] for encryption. The DES ECB mode is preferred because it is suitable for random data access. However, in this mode a given plaintext block always encrypts to the same ciphertext block. Therefore, a combination of ECB and OFB mode is used.

CFS requires the user to enter a passphrase (length usually greater than 16 characters). This passphrase is used to generate two keys, K_1 and K_2. K_1 is used to pre-compute a long pseudo-random bit mask (usually half a megabyte) using OFB mode. Encryption of a file is done block by block. Before encrypting, each block is first exclusive-or'd with a bit mask. To avoid identical blocks of a file encrypt to identical ciphertext, the bit mask chosen corresponds to the byte offset of the block modulo the mask length. However, using this scheme identical blocks at the byte offset present in different files will result in identical ciphertext. To avoid this, CFS XORs each cipher block of a file, with the corresponding inode. Since decryption requires the inode number used during encryption (which can change), CFS stores the inode number in the `gid` field of each file's inode. Decryption is the exact reverse of encryption. Pathnames and symbolic links are also encrypted in CFS. Access to the directories under `/crypt` is controlled by using UNIX file protection mechanisms. CFS also considers key recovery and secure key storage [13].

CFS is one of the first file system to provide a strong encryption scheme supporting block by block encryption and it instigated further research on cryptographic file systems. However, CFS is not completely transparent to the user. The granularity of encryption is a directory and the user should remember a key for all the encrypted directories. In addition, using a user-level NFS server reduces the performance of the system. CFS does not ensure integrity of the data and meta-data, and is cumbersome to use for group sharing as it does not include any key distribution techniques. Further, since all keys are dependent on passphrases that are associated with directories, in the case of emergency (password compromise) changing a passphrase for a directory should result in re-encryption of all the file located in that directory. However, to the best of our knowledge, CFS does not provide any such mechanism.

3.1.2 Other Systems

Numerous solutions exist that allow users to create secure partitions on their local disks and transparently encrypt data stored on the local disk [21, 34, 77, 80, 38, 62]. The main idea behind these systems is to create partitions (sometimes virtual) on the local disk and use passwords or pass phrases selected by users to generate a key for each partition. This key is then used to encrypt the data stored in the secure partitions. Gutmann's Secure FileSystem (GSFS [2]) for DOS [34] and Swank's SecureDrive [77] were one of the earliest of disk encrypting systems. GSFS provided encryption services for DOS and Windows. It allows users to create volumes on local disk that can be encrypted. Each volume appears as a normal DOS drive, but all of the data stored on it is encrypted at the individual-sector level. Volumes can be easily mounted or unmounted either manually or automatically after a certain time period. GSFS also provides other interesting services, such as threshold sharing to recover lost keys and using smart cards to store keys. SecureDrive is another device driver providing encryption services on DOS and Windows with somewhat lesser functionalities as compared to GSFS. The CryptoGraphic disk driver [21] provides

[2]Authors abbreviated it as SFS, but here it is called GSFS to avoid conflicts with other Secure File Systems (SFS).

encryption services on NetBSD. It consists of a kernel-level virtual disk driver that accesses the raw disk and performs block-level encryption and decryption. In addition, it consists of a user-level process that handles key generation. It supports four different key generation methods and the encryption algorithms can be selected by the user. The goal of disk encryption systems is to protect data stored on the local disk in order to reduce the risks of laptop thefts. Therefore, they are not designed for multi-user systems and typically the weakest link in the system is the password that is used to generate the keys.

Most of the systems use virtual memory; therefore, the data can still appear as plaintext in the unprotected virtual memory backing store. This problem was addressed by providing swap device encryption [64]. In order to protect data stored on the swap space (such as keys), it encrypts the swap device by randomly choosing keys. Keys are short-lived and are changed frequently. The data is automatically encrypted before it is stored on the swap space and it is automatically decrypted before the OS reads it from the swap space. However, this imposes a high performance penalty.

3.2 Shared Cryptographic File Systems

3.2.1 Transparent Cryptographic File System for UNIX

Overview TCFS [29] is an kernel-level file system that provides cryptographic services to the users. Similar to CFS, TCFS provides end-to-end security (encryption and decryption is performed at the client side). Compared to CFS, TCFS makes file encryption transparent to users, provides data integrity, and enables file sharing between a group (UNIX group) of users. File encryption is made transparent to the users by maintaining one bit information with each file that indicates whether the file is encrypted or not. Users have to maintain only one password that is used to encrypt all file-keys (rather than one password per encrypted directory as in the case of CFS). Users can select encryption algorithms that will be used by TCFS to encrypt file blocks in cipher-block chaining (CBC) mode.

Security Each user is associated with a *master-key* (created by the user), which is encrypted with the user's login password and stored at a central database. This database can be located on the client's machine or in a database located on a remote kerberized key server. Associated with every file is a randomly chosen *file-key*. The *file-key* is encrypted with the master-key and stored in the `file-key` field of the file header that is stored along with each file. A *block-key* is created per block of a file by hashing the result of the concatenation of the file-key and the block number. The block is then encrypted using the block key in the CBC mode. In addition to block encryption, an authentication tag, which is computed by hashing the concatenation of block data and block key is stored with the block. The readers can verify the integrity of the blocks by verifying the authentication code.

TCFS provides threshold sharing [73] to share files within a UNIX group. In this case, in order to acquire a group-key, a threshold number of share holders should be available on the same machine. A group-key is used to encrypt all the file-keys of the files belonging to that group. UNIX group is a special case of secret sharing where the number of secret shares is equal to one. To share files within a group, the system administrator should supply the threshold information, and the list of members of the group to the TCFS group creation utility. This program then generates a random key and encrypts each user's share with the user's password. To decrypt the group key, at least threshold number of users should login into the same machine and present their share to the kernel. This is inconvenient form of group sharing; however, this feature should be extended further for key recovery purposes (rather than key sharing). Since the utility that generates secret shares for the group members should have the users' password in clear, this becomes a serious vulnerability of the system. Further, the system administrator can acquire users' shares and use these shares to decrypt the group key. Therefore, in this case, TCFS does not protect data from the system administrators.

The master-key in TCFS is encrypted using the user's password. This becomes the weakest link of the system as users typically choose passwords from small domain. In the case of a password change, TCFS simply decrypts the master-key with the old password and re-encrypts with the new password. Therefore, if the user's password is revealed to the attacker, simply changing the master-key is not enough as the attacker could have downloaded all the keys using the compromised password. In this case, all the files (at least the ones that were updated) should be automatically decrypted using the old key and should be re-encrypted with fresh keys. Further, in the case of group sharing, the system administrator has to regenerate a new group key and redistribute the shares. It is not clear how TCFS handles these issues. In TCFS, the authentication tag is generated by hashing the concatenation of data and the block key. This is known to be insecure [50] and other secure constructions such as HMAC [11] should be used.

3.2.2 NCryptfs

Overview The NCryptfs [15] file system is a stackable file system designed to provide kernel-level encryption services. The main goal of NcryptFS is to provide transparent file encryption service that is easily portable without incurring significant performance overhead. The system administrator has to mount NCryptfs on `/mnt/ncryptfs`. The system administrator can attach one or more authorization entries with the NCryptfs mount point. An authorization entry specifies a salted hash of a password and associates some privileges to this hash. A user that can enter the password that matches a password in the authorization entry will be given the associated privileges (e.g., access permissions or delegation permissions).

Security NCryptfs grants access rights to an entity according to the authorization entry for that entity. Similar to CFS, in NCryptfs a user can *attach* a directory for encryption, which creates a directory entry in the NCryptfs mount point. The user is asked to enter a passphrase on every `attach` command, which is used to derive an encryption key. This encryption key is "pinned" in memory and is used for encrypting the files and file names stored under the corresponding attached directory. This key is used to encrypt page by page and supports multiple ciphers in CFB mode, where the initialization vector (IV) is a combination of inode number and page number. To share an attached directory with other users, the owner has to associate with the attached directory an authorization entry for each user. The users have to provide the owner with the salted hash of their password, which is stored by the owner in a configuration file. When the newly added user attempts to access a file stored in the specified attachment, NCryptfs uses the salted hash to authenticate the user. After successful authentication (by providing the password), NCryptfs uses the key generated during the attach process for encryption/decryption purposes. This key is provided during the attach operation; therefore, the owner has to be available (to attach the directory) before the user tries to access the shared files. Besides, since the key is stored in the kernel memory, the user and the owner have to access files from the same machine. Further, a user has to (remember and) supply to the owner a hash of his password for every directory he wishes to access. Therefore, NCryptfs is quite inconvenient to use for distributed file sharing.

Using the authorization entries, a owner can add arbitrary users and create his own adhoc group. Since these groups and their associated access rights can be different from those specified in the UNIX ACL of the file, NCryptFS allows users to bypass the VFS permissions. NCryptfs is quite efficient and portable due to its stackable kernel-level implementation. It does not ensure integrity of data and meta-data (inode information) and its key management for file sharing is very primitive. Since keys are passphrases, key recovery is left to the user. However, since keys are known only to the owners (as compared to TCFS), the system administrators cannot decrypt users' data (unless the passphrases are weak). Further, since the encryption keys are always stored in the kernel's memory, it is never revealed to other users (assuming that the kernel is secure). Therefore, revocation of users does not require re-encryption.

3.2.3 Encrypting File System (EFS) for Windows

Overview EFS [17, 18] provides cryptographic support to store Windows NT file system (NTFS) files encrypted on disk and on remote web shares. The goal of EFS is to provide transparent end-to-end file encryption service so that only legitimate clients can encrypt/decrypt these files. A variety of encryption algorithms can be used. In addition, EFS also enables file sharing between a small group of users and provides mechanisms to retrieve lost keys. However, EFS is not designed to ensure integrity of data and meta-data.

Security EFS automatically creates a public-private key pair for each user and acquires a certificate on the public key from the Certification Authority (CA) configured by the administrator. If a CA is not present, EFS self signs the public keys. File/directory encryption in EFS is performed using symmetric keys. If a user chooses to encrypt a file (or directory), EFS generates a file encryption key (FEK) for each file and encrypts the file using the encryption key. The FEK for that file is then encrypted using the user's public key and stored along with the encrypted file in a special EFS attribute called Data Decryption Field (DDF). If the file is shared by multiple users, the FEK will be encrypted by each user's (listed on the ACL) public key and the list of encrypted FEKs will be stored in DDF. Encrypting FEK for each individual user can incur a lot of overhead (especially for revocation) on the client side. Therefore, EFS is suitable for file sharing among a small number of users. The private key of each user can be stored securely in smart cards or in the integrated software-based protected store.

Whenever a user wants to access an encrypted file, EFS client automatically acquires the FEK for that file by decrypting the FEK using the private key of that user and then uses the FEK to decrypt/encrypt the file. These operations are transparent to the user. If remote files are stored on web folders, then the encryption and decryption operations are performed on the client's machine. However, if remote files are stored on the file shares, the encryption and decryption operations are performed on the computer on which the files are stored. Therefore, files are transferred in clear-text to the file share if the user is accessing files from another machine. Therefore, EFS is secure only if remote files are stored in a web folder.

EFS supports file recovery, that is, a user or user's organization can recover any encrypted file stored on the file system. A system administrator can define his recovery policy, which can restrict the recovery privileges only to legitimate users. A recovery policy can be configured at local, domain, and organization level. The system administrator creates one or more recovery agents and each recovery agent is assigned a public-private key pair. The file encryption key of each encrypted file is encrypted using the public key of each recovery agent and is stored along with the encrypted file in the Data Recovery Field (DRF) attribute. If a user looses his private key, he can send the encrypted file to one of the recovery agents who can then decrypt the file and send it back to the user.

Revocation is performed using Certificate Revocation List (CRL). Revoking a user's access rights to a file requires changing the access control list as well as removing the DDF entry for that user. EFS does not automatically change the FEK of that file and does not re-encrypt the file. Therefore, if the revoked user gets access physical access to the file, he can still decrypt the file and read all the contents. EFS does not provide this re-encryption service. EFS transparently encrypts data, but it does not provide integrity of data as well as confidentiality and integrity of meta-data. However, EFS has convenient and simple file sharing model as compared to TCFS and NCryptfs.

3.2.4 A Universal Access, Smart-Card-Based, Secure File System

Overview The smart-card based Secure File System [37] (SSFS [3]) enables clients to store data securely on local and remote sites using normal networking protocols. It allows secure file-sharing between two or more groups belonging to one organization or different organizations. In addition to encryption and distributed access control, SSFS also provides key recovery and secure key storage. All private keys (user and group private keys) are stored on smart-cards, which also performs all the cryptographic operations associated with those keys. Therefore, private keys never leave the smart-card; thus, keeping them secure. SSFS has three entities, namely users (producers and consumers of data), group servers, and file servers. A group server is maintained per group (or per project) and is completely trusted. Group servers make group membership and access control decisions for their respective groups. This ability enables project groups to define their group membership.

Security Every SSFS user and group server has a public-

[3]Authors abbreviated it as SFS, but here it is called SSFS to avoid conflicts with other Secure File Systems (SFS).

private key pair. Private keys are stored on personal smart cards. SSFS supports XML-based access control list. A owner can specify the users and groups that can decrypt his files in the access control list. Each file is encrypted with a symmetric key. To share files with individual users, the owner encrypts the file key with those user's public key. To share files with a group of users, the owner encrypts the file key with the corresponding group server's public key. Further, the owner can split the key between two or more group servers (or individuals) for key recovery purposes. In this case, the file encryption key can be recovered only if the specified number of share holders contribute their share.

Suppose Alice wants to share her file with the members of project Pentium. She encrypts the file key using the public key of Pentium's group server. Suppose Bob, a member of project Pentium wants to access Alice's file. Bob first acquires the file from the file server. After receiving the file, the SSFS client file system (on Bob's machine) examines the XML ACL to determine Bob's group server as it needs to acquire the file key to decrypt the file for Bob. The SSFS client then sends the encrypted key to the Pentium's group server along with a signed request. The signature is generated by Bob's smart card. After receiving the request from Bob, the Pentium group server authenticates Bob by verifying the Bob's signature and determines whether to grant or deny access to Bob based on the local policies. If access is granted, the group server takes the encrypted file-key, decrypts it, and then re-encrypts it to Bob's smart card (using Bob's public key). The group server then sends this re-encrypted key back to the SSFS client. The SSFS client sends the received encrypted key to Bob's smart card, which decrypts and returns the file-key. The file-key is used by the SSFS client to decrypt the file.

SFS attempts to address practical security issues in an organization by providing end-to-end encryption and key recovery. It does not restrict the group server to a fixed policy, instead the individual groups are free to use suitable policies. However, SSFS does not provide confidentiality of meta-data and integrity of data and meta-data. The SSFS group server is responsible for client authorization; therefore, it has to be online and can present a central point of failure. A SSFS user uses a smart-card in order to access the files. However, this requires smart-card support on all machines, which may not be practical for existing systems. Further, presence of the user level file system, smart-cards, communications with the group server, and public key operations during data path will result in high access latencies. Finally, it is not clear how write access control is performed by the file server (this can be complemented by capability based systems such as NASD).

3.2.5 SiRiUS

Overview SiRiUS [32] is a user-level file system designed to be layered over insecure network and peer-to-peer file systems such as NFS, CIFS, OceanStore, and Yahoo! Briefcase. It provides its own read-write cryptographic access control for file level sharing in small groups. The main goal was to design and implement a security mechanism that improves the security of a networked file system without making any changes to the file server. In addition to confidentiality of data, SiRiUS also ensures integrity of data, and loose integrity of meta-data.

Security SiRiUS is a user level file system implemented on Linux over NFSv3. All files are stored on the server in two parts, namely a data file ($d - file$), which contains the encrypted data and a meta data file ($md - file$), which contains the access control information. Storing access control information in a separate meta-data files allows SiRiUS to run on top of any storage server as long as the SiRiUS client can interact with the server according to the server's semantics. The $d - file$ is encrypted using a symmetric key and signed by the writer. Stored in each directory is the meta-data freshness file ($mdf - file$), which contains the root of the hash tree [51] of the $mdf - files$ associated with sub-directories. The root of the hash tree is signed periodically by the owner to ensure freshness of the meta-data. The SiRiUS client hides the presence of $md - file$ and $mdf - file$ to the user. It assumes the presence of some existing key distribution mechanism (e.g., PGP, S/MIME) to acquire users' authenticated public keys. All SiRiUS users have a public-private key pair. Associated with each file are two keys namely, an AES [50] File Encryption Key (FEK) and a DSA File Signing Key (FSK). FEK is used for encrypting the file and FSK is used for signing. Possession of FEK gives read only access to the file while the possession of both FEK and FSK bestows both read and write privileges to the beholder. Thus, the reader and writer separation is performed by giving appropriate keys. Files are encrypted using AES in counter mode [22].

To share files with other users the owner of the file creates an entry for every user in the $md - file$. Each entry contains the file's FEK encrypted with the public key of each user with read access. If the user also has a write access, then the entry for that user will also include the FSK of that file. The $md - file$ also contains the public key of FSK, the relative filename, hash of meta-data signed by the owner, and timestamp of last modification. The public key of FSK allows the readers to verify the signature on $d - file$, the relative filename prevents file-swapping attacks, and signature ensures integrity of meta-data.

Every directory contains a meta-data freshness file, which contains the root of the hash tree built from all the $md - files$ belonging to that directory and its subdirectories. The hash tree is built by hashing each $md - file$ with SHA-1 [50] and concatenating with the $md - files$ of each subdirectory. The owner's SiRiUS client will periodically time stamp the root $mdf - file$ and sign it using his private key. Verification can be performed by regenerating the $mdf - file$ for that directory and comparing with the current $mdf - file$. In order to perform read or writer operations on a file, a user acquires both the $md - file$ and the $d - file$, verifies signature on the $md - file$ (using public key of FSK), decrypts FEK and FSK (if the user is allowed to write), and performs read or write operations using the FEK. Key revocation is quite similar to file creation: the owner has to create new keys for every file that was accessible to the revoked user, encrypt the new keys with the public key of the non-revoked users, and sign the $md - file$ and the $mdf - files$.

SiRiUS is secure and provides secure file sharing with confidentiality and integrity of data. It does not consider privacy of meta-data. It ensures integrity of meta-data upto the time when the meta-data was last signed by the owner (since owner signs periodically). In addition, it prevents roll-back of $md - files$. All this is achieved without any modifications on the file server, a prudent design choice. Therefore, SiRiUS can be used on any of the existing file servers. If a user is revoked, all owner's have to scan through all their files looking for files that were accessible to the revoked user. In addition, all writers share the file signing key (FSK); therefore, it is impossible to trace (provide non-repudiation) the last person who modified a file. Further, owner has to re-encrypt the FEK (and FSK) for all existing user's, which will add performance overhead. Therefore, SiRiUS is good to use for small user groups. To reduce this performance overhead the author's have suggested to use the Naor-Naor-Lotspiech [56] construction.

3.2.6 Plutus

Overview Plutus [40], provides secure file sharing while placing minimal trust on the storage server. It enables end-to-end data and meta-data confidentiality, and data and meta-data integrity where all the key management and distribution is handled by the client. It provides an elegant revocation mechanism and reader-writer distinction similar to SiRiUS. In addition, it provides confidentiality and integrity of network messages (RPCs) sent between the client and the server. A prototype of Plutus is built on OpenAFS[58].

Security In Plutus, all files with identical sharing attributes are grouped in the same file-group. A file-group is a group of files with identical sharing attributes. This exploits the fact that even though a user typically owns and accesses many files, the number of equivalence classes of files with different sharing attributes is small; thus, enabling grouping of multiple files with same equivalence class. A simple example of file-group in UNIX is a group of all files owned by same owner and group having the same permission bits. A unique symmetric key called the $lockbox - key$ is associated with every file-group. A unique 3DES [50] symmetric key, called a $file - block\ key$ is used to encrypt each block of a file. A lockbox securely holds the keys for all the blocks of the files ($file - block\ keys$) belonging to one file-group by encrypting all the $file - block\ keys$ using the symmetric $lockbox - key$. If a user wants to share his files with other users, he creates a file-group (which contains permissions for the file-group and a list of members) and a $lockbox - key$, which has to be distributed by the creator of the file-group to the members of the file-group.

Each file is encrypted block by block using the respective $file - block\ key$, which is automatically created during block creation. The $file - block\ key$ is also automatically encrypted with the $lockbox - key$ of the associated file-group to share with other members. Plutus also encrypts each individual file name entry in the directory inode as well; thus, ensuring confidentiality of the data as well as meta-data. Associated with each filegroup (lockbox) is an RSA key pair where the private part of the pair is the $file - signkey$ and the public part is the $file - verifykey$. The readers are given the $lockbox - keys$ whereas the writers are given the $lockbox - keys$ as well as the $file - signkeys$. Thus, similar to SiRiUS, in Plutus possession of signing key distinguishes writers from readers. The file blocks are arranged in a Merkle tree [51] and the root is signed by writers using $file - signkey$ of that file-group, which can be verified by writers using the corresponding $file - verifykey$. This proves to the readers that the files are been modified by authorized writers and also ensures integrity of data as well as meta-data. The server authenticates the writers before allowing writes by using the write token associated per file-group, which is given by the owner to all writers of that file-group. This prevents someone from simply overwriting the entire storage space on the storage server.

Revocation in a secure file sharing system is expensive as it requires re-distribution of keys, re-encryption of the data accessible to the revoked user as well as re-signing of the revoked data. Plutus exploits the concept of lazy revocation first proposed in Cepheus [26]. Lazy revocation delays re-encryption until a file is updated. Another elegant feature of Plutus is key rotation. When a user is revoked, all his filegroup keys have to be changed by creating new filegroups. However, due to lazy revocation the existing (non-revoked) user of the revoked filegroup should have access to keys of both the revoked as well as the new file-group. In Plutus, the owner of the filegroup creates a new $lockbox - key$ for that filegroup by encrypting the current $lockbox - key$ using the owner's private key. The resulting key is the new $lockbox - key$. The members of the group can get the older version of $lockbox - key$ for that group by decrypting recursively using the owner's public key. The $file - signkeys$ and the $file - verifykeys$ are rotated in a similar fashion.

Plutus is a well designed secure file sharing system. Key distribution in Plutus is performed by the owner of the data. A user has to contact the owner to acquire keys, and, therefore, the owner has to be present online. To solve this problem, Plutus can be extended to use key distribution scheme like the one presented in Cepheus [26] (discussed in section 3.2.7). Finally, similar to SiRiUS, Plutus shares the $file - signkey$ with all the writers. Therefore, it is not possible to verify securely the last modifier of a file.

3.2.7 Other Shared Cryptographic File Systems

Cepheus [26] developed by Kevin Fu was one of the first secure file sharing systems. Plutus extends some of the concepts from Cepheus. The goal of Cepheus is to provide confidentiality and integrity of data and (part of) meta-data. Cepheus, introduces a group server that stores and delivers user's public keys and a encrypted group key associated with each UNIX group. A group key is created by the group owner and encrypted for each individual group member using the group member's public key. Each file block is encrypted with a unique symmetric key using RC5 in CBC mode. The owner of the file creates the file key while creating the file or while changing access control list due revocation of some user's rights. The file key is encrypted for the group using the group key and for the owner using the owners public key. These two versions of the encrypted file keys are placed in the file inode. To ensure file integrity, a writer also generates a keyed hash (HMAC) [50] on the root of the hash tree of the file blocks and the meta-data (using the file key). In order to access a file, a user first downloads the encrypted group key from the group server, decrypts the group key using his private key, fetches the encrypted file key from the inode and decrypts it using the group key, verifies the HMAC, and decrypts the file if the HMAC can be verified correctly. The group server authenticates the client by verifying his signature. Cepheus performs lazy encryption on revocation, an idea which was extended in Plutus. Cepheus does not provide reader-writer distinction, which as taken into consideration in Plutus.

SNAD is a secure file system that provides various integrity schemes that tradeoff security for performance. In

SNAD [53], clients encrypt every block of a file using RC5 encryption algorithm in CBC mode before sending a file to the storage server. The owner of the file creates a data structure called *key object* for an encrypted file, which stores a tuple for each user (or group) that is allowed to access that file. A tuple consists of a user ID (or group ID), the RC5 key encrypted with that user's (or groups) public key, and the permissions that should be granted to that user (or group). The key object also includes the ID and signature on the hash of the key object of the last writer to the key object. Every secure block contains the actual encrypted file block, the ID of the last writer, and either a writer's signature on the block or a HMAC (tradeoff of non-repudiation for performance) on the block using the writer's HMAC key. The HMAC key is stored on the device in another data structure called *certificate object*. It is not clear how the access to certificate object is controlled and how HMAC keys are added to the certificate object. The server either verifies the signatures of the user or the HMAC of the writer for every block before storing that block. Similarly, readers verify the signature or HMAC of the writer before reading a block. The key object can be modified by any person holding the appropriate file-key. Therefore, any user can add another user to the key object; thus, granting the new user access rights to files encrypted using that key object. SNAD trusts the storage server to perform access control. SNAD supports two additional schemes that improve the performance of the system; however, this tradeoff weakens security. In order to revoke a user, the file owner should re-encrypt the file with the new key and re-encrypt the key with each user's public key.

All of the secure file sharing approaches studied in this section require either the owner of the files or the owner of the group to securely share the file key or the group key with other user's. This is performed either by encrypting the appropriate key by using each user's public key or by using each user's passwords. While the latter approach is less secure and inconvenient the prior approach is expensive (especially for large groups) as the owner has to encrypt the keys using each user's public key. To reduce this computation overhead Ateniese et al. [28] proposed an interesting approach using *proxy based re-encryption* techniques. A high-level description of their approach is presented below. Suppose Alice desires to share her files with Bob. In proxy re-encryption, Alice creates a *token* for Bob and stores it with the storage server. The token is derived using Bob's public key and Alice's secret information. Alice encrypts the shared symmetric keys (e.g., file key) and stores them on the server. The storage server cannot decrypt these keys. The keys are encrypted by Alice in such a way that the storage server can re-encrypt them using Bob's token such that only Bob can decrypt them using his private key. Thus, the overhead of re-encryption is pushed to the server without revealing the keys to the server. Alice assigns such tokens for each user with whom the files are shared. While encrypting the file keys Alice specifies the tokens of the users that are allowed to decrypt the key. The storage server can transform (re-encrypt) the key only to the user's specified by Alice. Further, Alice does not have to online during the transformation process. This approach is secure and the only way a file's confidentiality can be compromised is if a user either publishes the symmetric key used for file encryption or the file itself.

4. STORAGE-BASED INTRUSION DETECTION

4.1 Self-Securing Storage

Self-Securing Storage System [75] is developed at CMU. It maintains old version of data for a specified time period so that system administrators can perform intrusion diagnosis and recovery using the history of old versions of the data. Self-securing storage runs a versioning system on the storage server side, which is independent of any client side software, such as the client operating system, or file system. It can also be running on the disk firmware itself. Since the versioning system is running on the server's hardware, it is independent of any client side compromises and cannot be disabled as long as the server is not compromised. S4, a prototype implementation, is available on Linux 2.2.14 Kernel.

The S4 storage server maintains a history pool of old data versions and an append-only audit log that logs information regarding all the commands sent to the server (e.g., the commands, the originator of the commands etc.). Instead of overwriting original data on every write command, the server creates a new version of the data and maintains both the versions. In addition, it runs a version cleaner that cleans (frees the space) the versions that are older than the duration of the time window for which the version history is to be maintained. The versions can be permanently deleted only by the cleaner program. This prevents accidental or on purpose deletion of the versions by a user or an attacker. The S4 storage server uses a log structure to maintain data an journal-based meta-data to provide efficient writes, reduce space utilization and reduce size of meta-data. A client interacts with the S4 server using S4 specific RPCs and a system administrator can interact with S4 server using the administrative interface provided by S4.

By using S4 on firmware of a storage drive one can securely maintain versions of object without huge tradeoff in terms of performance as well as disk capacity. Further, it will be very difficult to disable the versioning system running on the storage drive.

4.2 The CMU Storage-based IDS

The storage-based intrusion detection system (SIDS) [61] is based on the principal similar to that of the self-securing storage described above.

SIDS is embedded in a storage device and analyzes data access patterns and data modification characteristics, looking for manifestation of an attack. The main advantage of SIDS (running directly on the storage server or on the disk firmware) as compared to host-based IDS is that an intruder having full access to the host can disable a host-based IDS, whereas a SIDS can still continue to function properly and are independent of host (or OS) compromise. On the other hand, SIDS have a restricted view of the world as compared to host-based IDS and they also require some file-system level understanding, which restricts them to a particular FS. The CMU SIDS is a rule-based intrusion detection system (misuse detection) and uses a set of rules to detect suspicious modifications to data. A prototype implementation embedded in NFS server is provided. The implementation is quite efficient and proves that embedding SIDS into a storage device is quite feasible.

SIDS supports the detection of four categories of suspicious activities.

- Unexpected changes to important system files and binaries, using a rule-set very similar to Tripwire [44].
- Patterns of changes like non-append modification (e.g., of system log files) and reversing of inode times. These patterns indicate that attackers were trying to modify system log files with an intent of erasing evidence of their malicious activities.
- Specifically proscribed content changes to critical files (e.g., illegal shells inserted into /etc/passwd).
- Appearance of specific file names (e.g., hidden dot files) or content (e.g., known viruses or attack tools).

An administrative interface supplies the detection rules, which are checked by the server during the processing of each NFS request. Each request is analyzed to detect one of the four suspicious activities described above. When a detection rule triggers, the server sends the administrator an alert containing the full pathname of the modified file, the violated rule, and the offending NFS operation. It is assumed that a storage device or the storage server cannot be physically compromised and the administrative interface is secure. All of the malicious activities detected by SIDS can also be detected by an host-based IDS, however, host-based IDSs can be easily disabled by an intruder in control with the host OS. Further, the detection is performed for each RPC request, which gives a clear information about the files that are accessed. This can give a better result as compared to analyzing audit logs (such as UNIX system logs), which may not give all the necessary information.

4.3 Other Storage-based Intrusion Detection Systems

Several existing tools are designed to aid system administrators to monitor their file systems looking for manifestations of attacks. Tripwire [44] is one of the file integrity checkers that is widely used by both UNIX and Windows communities. It generates a signature (message digest) of monitored files using the original uninfected files and frequently checks the file stored in the system, looking for deviation from the recorded signatures. Several existing file integrity checks (e.g., Data Sentinel, Sentinel, Xintegrity [16]) work on a similar principle. Avfs [55] performs on-access antivirus checks. It is a stackable file system that intercepts filesystem reads and writes and scans for viruses, if necessary. By performing on-access anti-virus checks as compared to performing antivirus checks at open or close operations, Avfs reduces the window of attack and detects viruses before they are written to the disk. It extends ClamAV [1], an open source virus detection engine to improve the performance of pattern matching. After detecting a virus, Avfs quarantines the infected file (by denying access) or creates a new version of that file. The detection engine is stateful; therefore, in order to receive acceptable performance one needs to tradeoff the system's memory.

5. COMPARISON AND DISCUSSION

In this section we begin with comparing most of the systems presented above based on the security services provided by these systems. We have chosen a subset of the desired security services of a secure storage system that were discussed in the section 1.1. We compare the storage systems (listed in table 1) by taking a look at each security feature and discussing the advantages and disadvantages (if any) of the security mechanisms used by these systems. Further, we also raise potential research or implementation problems during the course of our discussion. Table 1 gives a high-level summarization of our comparison. The comparison is pertinent to the systems shown in the table. During comparison, the storage-based intrusion detection systems are not considered, as in order to have a fair comparison one will have to survey other IDS that are not specifically developed for storage yet can detect malicious activities on the storage servers.

Entity and Message Authentication: We have seen various flavors of entity authentication. Some systems maintain a separate authentication server while others leave this responsibility to the file server. Client authentication in NFS, AFS, SFS, TCFS is performed by a central trusted authentication server (AS) (Kerberos in the case of NFS and AFS). TCFS supports both password-based and Kerberos-based authentication. In NASD, the filemanager authenticates users and grants capability tokens to the users. The NASD storage devices authenticate the users based on these tokens. Cepheus and SSFS rely on a central group server to authenticate users. SFS provides two levels of authentication. At the first level, the SFS client machine mutually authenticates with the SFS file server. At the second level, the SFS user is authenticated by the SFS AS using the secure channel established during the first mutual authentication phase (file server forwards authentication requests to the AS). SFS uses the SRP [81] protocol that allows users to securely download their public-private keys from the password server without revealing any information about the their passwords and private keys to the server.

EFS and CFS rely on the underlying system's authentication mechanisms for this purpose. It is the job of the file server to verify the identity of the readers and the writers before allowing access to stored data. In SiRiUS and Plutus, writers sign before writing the data and readers verify the signatures before reading the data. The signing key is given to the writers by the owner of the data only after authenticating the writers. Since readers trust the owner, writers are implicitly trusted. In addition, in the case of Plutus, the file server verifies a write token given to writers by the owner before allowing writes on any data.

Message authentication involves authenticating the source of messages (such as RPC). Systems using Kerberos, provide mutual authentication between the client and the authentication server as well as between the client and the file server. Similarly NASD secures all messages and data sent between the client and the file server. In Plutus, both ends of the communicating parties share a symmetric key during RPC setup, which is used to HMAC all the RPC request and replies between them. SFS establishes a secure channel between the client and the file server.

Access Control: In SSFS access control is performed by the central group server. In the case of NASD, the central file manager gives a credential to the user that contains access rights on a particular object. The storage device simply enforces these rights. In the case of SiRiUS, access control is

Table 1: A comparison of security services provided by various storage systems. Auth., Cnfd., E2E, MD, Rec. and NA stand for authentication, confidentiality, end-to-end, meta-data, recovery and not applicable respectively. The Entity Auth. column describes how the client is authenticated by the authentication server (AS) and file server (FS). Message security indicates authentication, integrity, freshness, and privacy of messages (e.g., RPC) between the client and the file server. (1) These systems rely on the security mechanisms of the underlying file system for this feature. (2) EFS is not secure if files are stored in remote file shares. (3) Authentication is performed using digital signatures.

System	Entity Auth.		Access Control	Message Security	E2E Cnfd.		E2E Integrity		E2E Key Management			Revoke
	AS	FS			Data	MD	Data	MD	Grouping	Distribution	Rec.	
AFS	Kerberos		AFS ACL	Yes	No	No	No	No	NA	NA	NA	ACLs
NFSv4	Kerberos, LIPKEY		NT ACL	Yes	No	No	No	No	NA	NA	NA	ACLs
CFS	NA	No[1]	No[1]	No[1]	Yes	Yes	No	No	No	No file-sharing	Yes	No[1]
TCFS	Passwd	No[1]	No[1]	No[1]	Yes	No	Yes	No	Threshold	Admin encrypts group-keys using user's passwd	No	Immediate re-encrypt
EFS	No[1]	No[1]	No[1]	No[1]	Yes[2]	No	No	No	No	Owner encrypts file-key with each user's public key	Yes	Immediate re-encrypt
Cepheus	PKI[3]	No[1]	No[1]	No[1]	Yes	Yes	Yes	Yes	UNIX group	Group owner encrypts group-key with each user's public key	No	Lazy re-encrypt
Plutus	NA	write token	Read-write keys	Yes	Yes	Yes	Yes	Yes	File groups	Users contact owner for keys	No	Lazy re-encrypt
SiRiUS	NA	No[1]	Read-write keys	No[1]	Yes	Yes	Yes	Yes	No	File owner encrypts file-key with each user's public key	No	Immediate re-encrypt
SSFS	PKI[3]	No	AS	No[1]	Yes	No	No	No	Project	File owner encrypts file-key with each user's/ group's public key	Yes	Immediate re-encrypt
NASD	No[1]	CapKey	Capability	Yes	No	No	No	No	NA	NA	NA	Expiry, AV
SFS	PKI [3]	Credential from AS	SFS ACL	Yes	No	No	No	No	NA	NA	NA	CRL

completely based on possession of keys. If a user has the file encryption key, then he can read the data and if a user has file signing key, then he can modify the data. The file server does not perform access control. In Plutus, in addition to SiRiUS style access control mechanism, a file owner gives a write token to every writer. The Plutus file server verifies these tokens; therefore, the file server is partially trusted. In the remaining systems, access control is performed by the file server. Plutus, SiRiUS, and SSFS do not trust the file server to perform access control (the trust is partial in the case of Plutus), they assume that all file system data will be encrypted. However, this may not be a reasonable assumption as typically not all data is sensitive. Further, for the sake of performance a user many not want to encrypt all of the data. Therefore, the basic access control mechanisms have to be in place. Besides, at the least a file server should verify whether the user has write permissions (as done by Plutus) to an object, otherwise a malicious user can simply erase the entire disk. Therefore, the basic file system's access control mechanisms cannot be replaced and the file server has to be at least partially trusted to verify writes.

End-to-end Data and Meta-data Confidentiality: A file system supports end-to-end data and meta-data confidentiality if users of the data encrypt the data (and meta-data) before it is sent over the network and only users of the data (and not servers) decrypt the encrypted data. All except AFS, NFS, NASD, and SFS support end-to-end data confidentiality. AFS, NFS, SFS and NASD encrypt the network traffic between a client machine and a server. Cepheus, Plutus, and CFS support meta-data confidentiality.

Most of the real-world storage systems encrypt network traffic but do not support end-to-end confidentiality. The main reasons are reduced performance and the high amount of complexity involved in performing key management, key storage, backups, and revocation. Another important question is how can one support fast pattern searching (such as `grep`) on the encrypted data? Modern storage servers have content based indexing capabilities for providing fast content based search and retrieval. If the data is encrypted by the user, the storage server will not have the data in clear, which will prevent them from performing content based search and indexing. Is it possible to allow the storage server to build content based indexes on the encrypted data without revealing any information to the server about the plaintext? It might be possible to build some encrypted keywords on the client side before the client writes the file to the storage server. However, this may not give enough information to the server. This is an active area of research [33, 14] and hopefully we will see fast and practical pattern matching techniques on encrypted data.

Further, what should the file system do if an encrypted file is transferred to another file system that does not support encryption/decryption services? EFS, for example, decrypts a file before transferring it to a different file system. While this makes the encrypted file accessible to the user, this also defeats the purpose of encryption. Transferring the file encrypted and leaving the decryption operations to user (who may not be aware of the underlying encrypted file system) will cause inconvenience to the user. These issues should be resolved to make end-to-end encryption useable in practice.

End-to-end key management: Grouping is typically used to reduce the key management burden. In Cepheus, a group is same as a UNIX group whereas Plutus groups files with same owner, group, and access rights into one file-group. The grouping mechanism used in Plutus reduces the number of keys as typically a UNIX system has lesser file-groups as compared to UNIX groups. SSFS groups users according to projects within an organization. TCFS provides threshold group sharing where key used for encryption is split between a group of members.

One important aspect of key management is key distribution. Key distribution is performed in-band or out-of-band. In the case of in-band key distribution file-keys are stored encrypted along with the files. For example, SiRiUS, EFS, SSFS store file-keys encrypted with each user's public key along with the meta-data of the file. Out-of-band key distribution requires key distribution to be handled by some other mechanism. For example, in Plutus readers and writers have to contact the group owners to acquire appropriate keys.

Key recovery is another essential feature that should be provided in storage systems that support encryption and signing. Key recovery is useful for recovering lost keys and recovering old backed-up encrypted data even in the absence of the encryptors. EFS, CFS, and SSFS provide key recovery. Key recovery should be inherent (mandatory) to the system (as in the case of EFS) and not discretionary as a user can simply ignore the key recovery process. Many key recovery systems are already been proposed. Identifying appropriate key recovery mechanisms and integrating them with the secure file systems (or key creation programs) is important.

Revocation: In the case of encrypting file system, if a user is revoked, all the files accessible to the revoked user may need to be re-encrypted as the user can store the keys and attempt to get physical access to the data. Cepheus and Plutus perform lazy revocation where the owner marks the files (accessible to the revoked user) as revoked and the file is re-encrypted on the next update to that file. In the case of NFS and AFS (or any non-encrypting file system) revocation is performed by changing the file ACL. NASD uses expiry and access control version for revocation. Systems that use public key certificates (such as EFS, SFS, SSFS) have to maintain servers to distribute certificates and CRLs.

Non-repudiation: It is important, in some commercial environments, to ensure non-repudiation of writes; that is, to prevent writers from denying their modifications to the data. While this can be easily achieved by digital signatures (albeit with small performance overhead), except SNAD none of the existing systems provide non-repudiation of writes/updates.

Key Storage: In addition to file encryption secure storage of keys is also important. CFS and SSFS use smart-card to store keys, and, therefore, the keys are stored securely without trusting any other central entity for this purpose. Systems such as Cepheus and TCFS encrypt users keys using user's password. In this approach, keys are as secure as a user's password. However, considering user's convenience this is a practical approach as most of the systems today rely on password-based authentication. A related question is should we change keys (and hence re-encrypt all the data) when a user changes his password? It is difficult to distinguish whether a user is attempting to change his password because of a compromise or as a part of a routine password change. Further, sometimes system administrators create passwords for user's. For example, if the user is a new user to the system, or if a user forgets his password. In this case, the system administrator can access all the files. How can one resolve this problem?

Long Term Key Management: Keys should be securely managed as long as the data is in existence. This will raise many management issues as a lot of unforeseen changes can happen during the lifetime of the data. For example, the user who encrypted the data as well as system administrators may not be available at the time when the data is required to be decrypted. How can the person who may be unknown to the system at the time of decryption efficiently acquire the necessary decryption keys or verification keys? Further, keys can compromised or cryptographic algorithms can be considered compromised (or considered weak) in the future. As suggested in [10] re-encryption of the data on the storage server side can solve this problem. However, the focus of the problem described above now shifts towards securely managing keys that were used by the server for re-encryption. Therefore, practical and secure management approaches should be proposed and standardized.

6. CONCLUSION

In this paper, we have presented a comprehensive survey of existing secure storage systems. We have listed the basic security services that should be provided by an ideal secure storage system. However, it is evident that for all practical purposes it is difficult to build a storage system that can satisfy all the listed services. However, this list can help system designers to quantify their requirements and evaluate other storage systems. We have categorized the existing solutions and presented case-study of each category. Finally, we have presented comparison of most of the surveyed systems and raised a few practical questions that should be addressed by a secure storage system.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (NSF) under Grant CNS-0448423 and by the Intelligent Storage Consortium at the Digital Technology Center (DTC), University of Minnesota.

8. REFERENCES

[1] The clam antivirus toolkit. `http://www.clamav.net`.

[2] The Directive 2002/58/EC of the European Parliament and of the Council. `http://europa.eu.int/eur-lex/pri/en/oj/dat/2002/l_201/l_20120020731en00%370047.pdf`.

[3] The Health Insurance Portability and Accountability Act. `http://www.hipaa.org/`.

[4] The IEEE Security in Storage Working Group. `http://siswg.org`.

[5] Design Criteria Standard for Electronic Records Management Software Applications (DoD 5015.2-STD). `http://www.dtic.mil/whs/directives/corres/html/50152std.htm`, 2002.

[6] The Sarbanes-Oxley Act of 2002. `http://news.findlaw.com/hdocs/docs/gwbush/sarbanesoxley072302.pdf`, 2002.

[7] K. S. Amiri. *Scalable and Manageable Storage Systems.* PhD thesis, CMU, December 2000.

[8] S. Axelsson. Intrusion detection systems: A survey and taxonomy. White Paper, Chalmers University, Sweden, 2000.

[9] A. Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, and J. Satran. A two layered approach for securing an object store network. In *SISW*, December 2002.

[10] D. Beaver. Network security and storage security: Symmetries and symmetry-breaking. In *Proceedings of the IEEE Security In Storage Workshop*, 2002.

[11] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash function for message authentication. Proceedings of CRYPTO, 1996.

[12] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Communications and Computing Security.*

[13] M. Blaze. Key management in an encrypting file system. In *Proceedings of Summer 1994 USENIX Technical Conference.*

[14] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano. Public-key encryption with keyword search. In *Proceedings of Eurocrypt*, 2004.

[15] E. Z. Charles P. Wright, Michael C. Martino. Ncryptfs: A secure and convenient cryptographic file system. In *USENIX Annual Technical Conference*, June 2003.

[16] File integrity checkers. `http://www.serverfiles.com/Network-security-software/File-integrity-checkers/date/`.

[17] M. Corporation. Encrypting File System for Windows 2000. White Paper, July 1999.

[18] M. Corporation. Encrypting file system in Windows XP and Windows Server 2003, August 2002. `http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/cryptfs.ms%px`.

[19] D. E. Denning, M. Bellare, S. Goldwasser, and E. R. Verheul. *Descriptions of Key Escrow Systems.* `http://www.cosc.georgetown.edu/~denning/crypto/Appendix.html`, February 1997.

[20] D. E. Denning and D. K. Branstad. A taxonomy for key escrow encryption systems. *Communications of the ACM*, 39(3), 1996.

[21] R. C. Dowdeswell and J. Ioannidis. The CryptoGraphic Disk Driver. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, June 2003.

[22] M. Dworkin. Recommendation for block cipher modes of operation. In *Special Publication 800-38A, NIST*, 2001.

[23] M. Eisler. LIPKEY - a low infrastructure public key mechanism using SPKM. RFC 2847, June 2000.

[24] M. Eisler, A. Chiu, and L. Ling. RPCSEC_GSS protocol specification. RFC 2203, September 1997.

[25] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft (draft-freier-ssl-version3-02.txt), Networking Group, March 1996. `http://wp.netscape.com/eng/ssl3/ssl-toc.html`.

[26] K. Fu. Group sharing and random access in cryptographic storage file system. Master's thesis, MIT, 1999 June.

[27] K. Fu, F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *Proceedings ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[28] M. G. S. H. Giuseppe Ateniese, Kevin Fu. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[29] A. D. S. Giuseppe Cattaneo, Luigi Catuogno and P. Persiano. Design and implementation of a transparent cryptographic file system for UNIX. In *FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[30] H. Gobiof. *Security for high performance commodity subsystem.* PhD thesis, CMU, July 1999.

[31] H. Gobioff, G. Gibson, and D. Tygar. Security for network attached storage devices. Technical report, Carnegie Mellon University, 1997.

[32] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed Systems Security (NDSS) Symposium*, pages 131–145, February 2003.

[33] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/`.

[34] P. Gutmann. Secure FileSystem (SFS), September 1996. `http://www.cs.auckland.ac.nz/~pgut001/sfs`.

[35] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[36] J. Howard. An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, Dallas, TX, February 1998.

[37] C. F. J. Hughes. Architecture of the secure file system. In *Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems*, April 2001.

[38] Jetico. BestCrypt. `http://www.jetico.com/download.htm`.

[39] J. Kaiser and M. Reichenbach. Evaluating security tools towards usable security: A usability taxonomy for the evaluation of security tools based on a categorization of user errors. Technical report, Usability, 2002.

[40] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus — scalable secure file sharing on untrusted storage. In *USENIX File and Storage Technologies (FAST)*, 2003.

[41] M. Kaminsky, G. Savvides, D. Mazieŕes, and M. F. Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[42] A. Kent and R. Atkinson. Security architecture for the Internet protocol. RFC 2401, Network Working Group, November 1998. `http://www.ietf.org/rfc/rfc2401.txt`.

[43] V. Kher and Y. Kim. Decentralized authentication mechanisms for object-based storage devices. In *IEEE SISW*, October 2003.

[44] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, 1994.

[45] J. Kubiatowicz, D. Bindel, Y. Chen, 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. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[46] J. Li, M. Krohn, D. Mazires, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, December 2004.

[47] J. Linn. Generic security service application program interface. Request for Comments 1508, September 1993.

[48] J. Linn. The kerberos version 5 GSS-API mechanism. RFC 1964, June 1996.

[49] D. Mazières. *Self-certifying file system*. PhD thesis, MIT, May 2000.

[50] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[51] R. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[52] S. Microsystems. RPC: Remote procedure call. Request for Comments 1050, April 1998.

[53] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proceedings of the Conference on File and Storage Technologies (FAST 2002)*, pages 1–13, January 2002.

[54] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. Keromytis, and J. Smith. Secure and flexible global file sharing. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track*, 2003.

[55] Y. Miretskiy, A. Das, C. Wright, and E. Zadok. Avfs: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[56] D. Naor, M. Naor, and J. Lotspiech. Revocation and tracing schemes for stateless receivers. In *Proceedings of Crypto*, volume 2139, pages 41–62, August 2001.

[57] B. C. Neumann and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[58] OpenAFS, open source version of AFS. `http://www.openafs.org`.

[59] Information technology - SCSI Object-Based Storage Device Commands -2 (OSD-2). T10 Working Draft, October 2004. `http://www.t10.org/ftp/t10/drafts/osd2/osd2r00.pdf`.

[60] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. SANE 2000, May 2000.

[61] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *12th USENIX Security Symposium*, Washington, D.C., August 2003.

[62] PGPdisk. `http://www.pgpi.org/products/pgpdisk/`.

[63] Draft standard for information technology - portable operating system interface (POSIX) - amendment: Protection, audit and control interfaces. Portable Applications Standards Committee (PASC), November 1992.

[64] N. Provos. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[65] T. Rabin. A simplified approach to threshold and proactive RSA. In *Proceedings of Crypto*, 1998.

[66] B. Reed, E. Chron, R. Burns, and D. D. E. Long. Authenticating network attached storage. *IEEE Micro*, 20(1):49–57, January 2000.

[67] B. C. Reed, M. A. Smith, and D. Diklic. Security considerations when designing a distributed file system using object storage devices. In *SISW*, December 2002.

[68] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the USENIX File and Storage Technologies Conference (FAST)*, 2003.

[69] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the Conference on File and Storage Technology*, January 2002.

[70] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[71] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[72] M. Satyanarayanan. A survey of distributed file systems. Annual Review of Computer Science, 1990.

[73] A. Shamir. How to share a secret. *Comm. ACM*, 24(11), November 1979.

[74] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3530, April 2003.

[75] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing storage: Protecting data in compromised systems. In *OSDI*, October 2000.

[76] The NFS distributed file service. A White Paper from SunSoft, November 1995.

[77] E. Swank. SecureDrive. `http://www.stack.nl/~galactus/remailers/securedrive.html`.

[78] B. Taylor and D. Goldberg. Secure networking in the sun environment. In *Proceedings of the Summer Usenix Conference*, 1986.

[79] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *8th USENIX Security Symposium*, 1999.

[80] S. Winter. Sentry. `http://www.softwinter.com/documentation/nt/sentry_2.htm`.

[81] T. Wu. The secure remote password protocol. In *Proceedings of the Network and Distributed System Security Symposium*, 1998.