RZ 3661 (# 99681) 09/11/06
Computer Science 18 pages

Research Report

Cryptographic Security for a High-Performance Distributed File System

Roman Pletka and Christian Cachin

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{rap, cca}@zurich.ibm.com

Cryptographic Security for a High-Performance Distributed File System

Roman Pletka Christian Cachin

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{rap,cca}@zurich.ibm.com

10 September 2006

Abstract

Storage systems are increasingly subject to attacks. Cryptographic file systems mitigate the danger of exposing data by using encryption and integrity protection methods and guarantee end-to-end security for their clients. This paper describes a generic design for cryptographic file systems and its realization in a distributed storage-area network (SAN) file system. Key management is integrated with the meta-data service of the SAN file system. The implementation supports file encryption and integrity protection through hash trees. Both techniques have been implemented in the client file system driver. Benchmarks demonstrate that the overhead is noticeable for some artificially constructed use cases, but that it is very small for typical file system applications.

1 Introduction

Security is quickly becoming a mandatory feature of data storage systems. Today, storage space is typically provided by complex networked systems. These networks have traditionally been confined to data centers in physically secured locations. But with the availability of high-speed LANs and storage networking protocols such as FCIP [27] and iSCSI [29], these networks are becoming virtualized and open to access from user machines. Hence, clients may access the storage devices directly, and the existing static security methods no longer make sense. New security mechanisms are required for protecting stored data in virtualized and networked storage systems.

In distributed storage systems, data exists in two different forms, leading also to different exposures to unauthorized access:

Data in flight: Data that is in transit on a network, between clients, servers, and storage devices. Unauthorized access may occur from other nodes on the network. These attacks and their countermeasures are similar to the situation for other communication channels, for which cryptographic protection is widely available.

Data at rest: Data that resides on a storage device. An attacker may physically access the storage device or send appropriate commands over the network. If the network is not secure, these commands may also be initiated by clients that are authorized to access other parts of the storage system. Data at rest differs from data in flight on a communication channel because it must be accessible in arbitrary order, and not only read in the order it was written. Hence, new cryptographic methods are needed for protecting it.

Data at rest is generally considered to be at higher risk than data in flight, because an attacker has more time and flexibility to access it. Moreover, new regulations such as Sarbanes-Oxley, HIPAA, and Basel II also dictate the use of encryption for data at rest.

A secure storage system should guarantee the following security properties:

Confidentiality: Protection against unauthorized disclosure of data, for example, through eavesdropping on the network.

Integrity: Protection against unauthorized modification of data, for example, by making subtle changes to stored data.

Storage systems use a layered architecture, and cryptographic protection can be applied on any layer. For example, one popular approach used today is to encrypt data at the level of the block-storage device, either in the storage device itself, by an appliance on the storage network [14], or by a virtual device driver in the operating system (e.g., Loopback encryption in Linux [16]). The advantage is that file systems can use the encrypted devices without modifications, but the drawback is that such file systems cannot extend the cryptographic security to the users. The reason is that any file-system client can access the storage space in its unprotected form, and that access control and key administration take place below the file system.

In this paper, we address encryption at the file-system level. We describe the design and implementation of cryptographic protection methods in a high-performance distributed file system. After introducing a generic model for secure file systems, we show how it can be implemented using SAN.FS, a SAN file system from IBM [25]. Our design addresses confidentiality protection by data encryption and integrity protection by means of hash trees. A key part of this paper is the discussion of the implementation and an evaluation of its performance. The model itself as well as our design choices are generic and can be applied to other distributed file systems.

Encryption in the file system maintains the end-to-end principle in the sense that stored data is protected at the level of the file-system users, and not at the infrastructure level, as is the case with block-level encryption for data at rest and storage-network encryption for data in flight. Moreover, an optimally secure distributed storage architecture should minimize the use of cryptographic operations and avoid unnecessary decryption and re-encryption of data as long as the data does not leave the file system. This can be achieved by performing encryption and integrity protection of data directly on the clients in the file system, thereby eliminating the need to separately encrypt the data in flight between clients and storage devices. Given the processing capacity of typical workstations today, encryption and integrity verification add only a small overhead to the cost of file-system operations, as our benchmarks demonstrate.

Distributed file systems like SAN.FS and cluster file systems are usually optimized for performance, capacity, and reliability. For example, in SAN.FS and in the recent pNFS effort [15], meta-data operations are separated from the data path for increasing scalability. From a security perspective, such an approach might sometimes be suboptimal or even make it impossible to provide end-to-end security. This work shows that cryptographic security can be added to high-performance distributed file systems at minimal additional cost.

The remainder of this paper is organized as follows. Section 2 introduces a general model for secure file systems and discusses related work. Then, Section 3 describes the design of SAN.FS and how cryptographic extensions can be added to it. Section 4 provides more details about our implementation of cryptographic extensions to SAN.FS, which are also applicable to other distributed file systems. Finally, Section 5 shows our performance results. A discussion of our approach with respect to related work concludes the paper.

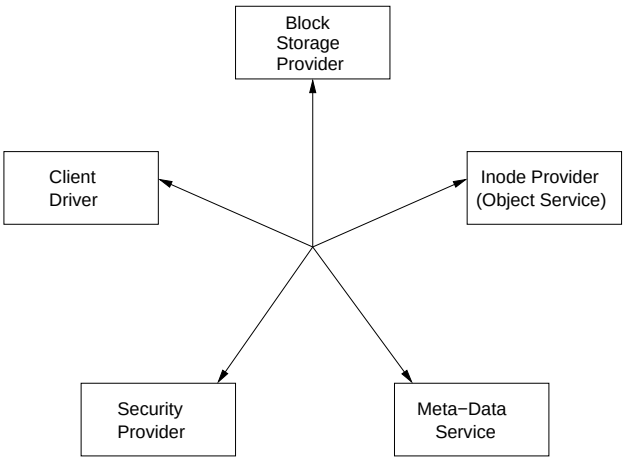


Figure 1: Components of a distributed file system.

2 Model and Related Work

This section first presents an abstract model of a distributed file system, introduces cryptographic distributed file systems, and reviews previous work in the area.

2.1 File System Components

File systems are complex programs designed for storing data on persistent storage devices such as disks. A file system manages the space available on the storage devices, provides the abstraction of files, which are data containers that can grow or shrink and have a name and other meta-data associated to them, and manages the files by organizing them into a hierarchical directory structure.

Internally, most file systems distinguish at least the following five components as shown in Figure 1: (1) a *block-storage provider* that serves as a bulk data store and operates only on fixed-size blocks; (2) an *inode provider* (or *object-storage service*), which provides a flat space of storage containers of variable size; (3) a *meta-data service*, handling abstractions such as directories and file attributes and coordinating concurrent data access; (4) a *security provider* responsible for security and access-control features; and (5) a *client driver* that uses all other components to realize the file system abstraction to the operating system on the client machine.

The first three components correspond to the layered design of typical file systems, i.e., data written to disk in a file system traverses the file-system layer, the object layer, and the block layer in that order. The security provider is usually needed by all three layers. In most modern operating systems, the block-storage provider is implemented as a block device in the operating system, and therefore not part of the file system.

In traditional file systems, all components reside on the same host in one module. With the advent of high-speed networks, it has become feasible to integrate file system components across several machines into distributed file systems, which allow concurrent access to the data. A network can be inserted between any or all of the components, in principle, and the networks themselves can be shared. For example, in storage-area networks only the storage provider is accessed over a network; in distributed file systems such as NFS and AFS, the client uses a network to access a file server, which contains storage, inode, and meta-data providers. The security provider can be an independent entity in AFS and in NFSv4.

The NASD architecture [9] and its successor Object Store [2] propose network access to the object-storage service. Compared with accessing a block-storage provider over the network, this design simplifies the security architecture. The security model for object storage [1] assumes that the device is trusted to enforce access control on a per-object basis. The security provider is realized as an independent entity, accessed over a network. Object storage is an emerging technology, and, to our knowledge, distributed file systems in which clients directly access object-storage devices are not yet widely available.

In SAN.FS, on which we focus in the remainder of this paper, clients access the storage devices directly over a SAN (i.e., using Fibre Channel or iSCSI). All meta-data operations are delegated to a dedicated server, which is accessed using TCP/IP over a local-area network (LAN).

2.2 Cryptographic File Systems

Cryptographic file systems encrypt and/or protect the integrity of the stored data using encryption and data authentication. Cryptography is used because the underlying storage provider is not trusted to prevent unauthorized access to the data. For example, the storage provider may use removable media or must be accessed over a network, and therefore proper access control cannot be enforced; another common example of side-channels to the data are broken disks that are being replaced.

In a system using encryption, access to the keys gives access to the data. Therefore, it is important that the *security provider* manages the encryption keys for the file system. Introducing a separate key management service, which has to be synchronized with the security provider providing access control information, only complicates matters. Analogously, the security provider should be responsible for managing integrity reference values.

Cryptographic file systems exist in two forms: either as an enhancement within an existing physical file system that uses an underlying block-storage provider, or as a virtual file system that must be mounted over another (virtual or physical) file system. The first approach results in *monolithic* cryptographic file systems that can be optimized for performance. The second approach results in *stackable* or *layered* file systems [33], whose advantage lies in the isolation of the encryption functionality from the details of a physical file system. In this way, the encryption layer can be reused for many physical file systems. But because multiple copies of the data must be maintained by the operating system, stackable file systems are generally slower than monolithic ones.

2.3 Previous Work on Cryptographic File Systems

A considerable number of prototype and production cryptographic file systems have been developed in the past 15 years. We refer to the excellent surveys by Wright *et al.* [32] and by Kher and Kim [20] for more details, and mention only the most important systems here.

Most early cryptographic file systems are layered and use the NFS protocol for accessing a lower-layer file system: CFS [3] uses an NFS loopback server in user space and provides per-directory keys that are derived from passphrases; TCFS [5] uses a modified NFS client in the kernel and utilizes a hierarchical key management scheme, in which per-user master keys to protect per-file keys are maintained. SFS [22, 23, 24] is a distributed cryptographic file system also using the NFS interfaces, which is available for several Unix variants. These systems do not contain an explicit security provider responsible for key management, and delegate most of that work to the user.

SUNDR [21] is a distributed file system that works with a completely untrusted storage server. It uses a content-addressable block store and provides a fork-consistent view of the file system to the clients, which guarantees that clients can detect any violation of integrity and consistency, as long as they see file updates of each other. SUNDR provides file encryption and integrity protection using hash trees, but makes heavy use of digital signatures.

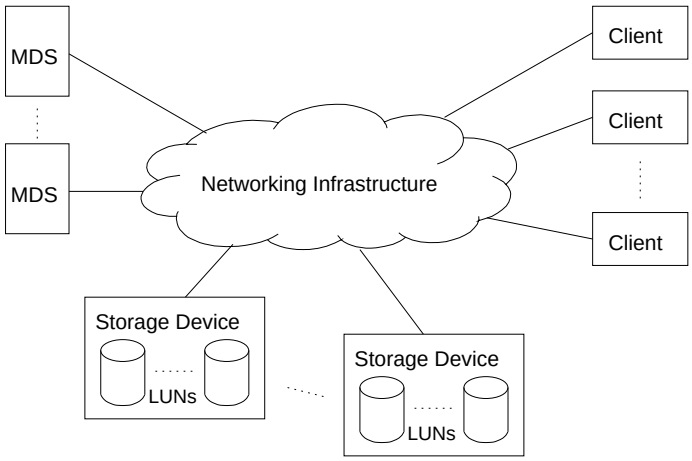


Figure 2: The architecture of SAN.FS.

Microsoft Windows 2000 and later editions contain an extension of NTFS called EFS [28], which provides file encryption with shared and group access. It relies on the security provider in the Windows operating system for user authentication and key management. As it is built into NTFS, it represents a monolithic solution.

Some more recent cryptographic file systems follow the layered approach: NCryptfs [31] and eCryptFS [12, 13] are two native Linux file systems, which are implemented in the kernel and use stacking at the VFS layer based on the FiST framework [34]. EncFS [11] for Linux is implemented in user-space relying on Linux's file system in user space module (FUSE). FUSE intercepts system calls at the VFS layer and redirects them to the daemon in user space. NCryptfs, eCryptFS, and EncFS currently provide only manual key management on a per-filesystem basis, but the eCryptFS design includes support for a sophisticated key management scheme with per-file encryption keys and shared access using public-key cryptography.

Except for Windows EFS and apart from using a stackable file system on top of a networked file system such as NFS or AFS, there are currently no distributed cryptographic file systems that allow file sharing and concurrent access to encrypted files.

All file systems mentioned support encryption, but only few of them also provide data integrity through hash functions or digital signatures.

3 Design

This section presents the SAN File System (SAN.FS) and our design for turning SAN.FS into a cryptographic file system supporting confidentiality and integrity.

3.1 SAN.FS

SAN File System (SAN.FS) from IBM, also known as *Storage Tank*, implements a distributed file system on a SAN, providing shared access to virtualized storage devices for a large heterogeneous set of clients, combined with policy-based file allocation [25]. It is scalable because the clients access the storage devices directly over the SAN. This is achieved by separating meta-data operations from the data path and by breaking up the traditional client-server architecture into three components, as shown in Figure 2.

The three components of SAN.FS are the following: First, a client driver, which comes in several variations, as a VFS provider for Unix-style operating systems such as Linux and AIX, or as an installable file system for Microsoft Windows. The client driver also implements an object service (according to Section 2.1) as an intermediate layer. Second, the meta-data server (MDS), which runs on a dedicated cluster of nodes, implements all meta-data service abstractions such as directories and file meta-data, and performs lock administration for file sharing. Third, the storage devices, which are standard SAN-attached storage servers that implement a block-storage service. Note that SAN.FS does not contain a security provider, but delegates this function to the clients.

In SAN.FS, all bulk data traffic flows directly between a client and the storage devices over the SAN. The client communicates with the MDS over a LAN using TCP/IP for allocating storage space, locating data on the SAN, performing meta-data operations, and coordinating concurrent file access. The protocol between the client and the MDS is known as the *SAN.FS protocol* and publicly available [17]. The MDS is responsible for data layout on the storage devices. It also implements a distributed locking protocol in which leases are given to clients for performing operations on the data [4]. As the clients heavily rely on local data caching to boost performance, the MDS essentially implements a cache controller for all clients in SAN.FS.

SAN.FS maintains access control information such as file access permissions for Unix and the security descriptor for Windows in the meta-data, but leaves its interpretation up to the client operating system [17]. In order to implement proper access control for all users of a SAN.FS installation, one must therefore ensure that only trusted client machines connect to the MDS and to the SAN. It is possible to share files between Windows and Unix.

3.2 Cryptographic SAN.FS

The goal of our cryptographic SAN.FS design is to provide end-to-end confidentiality and integrity protection for the data stored by the users on the SAN.FS clients where all cryptographic operations occur only once in the data path. We assume that the MDS is trusted to maintain cryptographic keys for encryption and reference values for integrity protection, and neither discloses them to, nor accepts modifications to them, from unauthorized clients. We also assume that the clients properly enforce file access control. Storage devices and other entities with access to the SAN are untrusted entities that potentially attempt to violate the security policy. Hence, using the terminology of Section 2.1, the meta-data provider also implements the security provider.

Corresponding with the design goals of SAN.FS, the client also performs the cryptographic operations and sends the protected data over the SAN to the storage devices. Encryption keys and integrity reference values are stored by the MDS as extensions of the file meta-data. The links between clients and the MDS are protected using, e.g., IPsec or Kerberos. The encryption and integrity protection methods are described later in this section.

A guideline for our design was to leave the storage devices unmodified. This considerably simplifies deployment with the existing, standardized storage devices without incurring additional performance degradation. But a malicious device with access to the SAN can destroy stored data by overwriting it, because the storage devices are not capable of checking access permissions. Cryptographic integrity protection in the file system can detect such modifications, but not prevent them. Therefore, integrity failure events need to be reported to the administrator, who can initiate an adequate recovery process, e.g., restore data from backup.

We remark that an alternative model for implementing strong access control in storage devices is available with object storage [1, 2]. It prevents any unauthorized modification to the data by other nodes on the SAN. Our design is orthogonal to the security design of object storage, and could easily be integrated in a SAN file system using object-storage devices.

3.2.1 Confidentiality Protection

The confidentiality protection mechanism encrypts the data to be stored on the clients with a symmetric cryptosystem, using a per-file encryption key. Each disk-level data block is encrypted with the AES block cipher in CBC mode, with an initialization vector derived from the file object identifier and from the offset of the block in the file and the per-file key. These choices ensure that all initialization vectors are distinct.

Instead of CBC mode, it would also be possible to use a tweakable encryption mode, such as those being considered for standardization in the IEEE P1619 effort [18]. These modes offer somewhat better protection against active attacks on the stored data, because even a small change to an encrypted block will cause the recovered plaintext to look random and completely independent of the original plaintext. With CBC mode, an attacker can have some influence on the recovered plaintext, when no additional integrity protection method is used. Despite this deficiency, we chose CBC mode because it offers better performance (essentially twice the speed, because it is implemented in software) and because our integrity protection scheme provides complete defense against modifications to the stored data.

The file encryption key is unique to every file and stored as part of a file's meta-data. As such a key is short (typically 16–32 bytes), the MDS can easily be changed to accommodate it. The key can be chosen by either the MDS or the client.

3.2.2 Integrity Protection

The integrity protection mechanism detects unauthorized modification of data at rest or data in flight by keeping a cryptographic hash or "digest" of every file. The hash value is short, typically 20–64 bytes with the SHA family of hash function [6], and is stored together with the file meta-data by the MDS. All clients writing to the file also update the hash value at the MDS, and clients reading file data verify that any data read from storage matches the hash value obtained from the MDS. An error is reported if the data does not match the hash value.

The hash function is not applied to the complete file at once, because the hash value would have to be recomputed from scratch whenever only a part of the file changes, and data could only be verified after reading the entire file. This would incur a prohibitive overhead for large files. It is important to use a data structure that allows verification and manipulation of hash values with an effort that is roughly proportional to the amount of data affected.

The well-known solution to this problem is to create a *hash tree*, also known as *Merkle tree* [26], and to store it together with the file. A hash tree is computed from the file data by applying the hash function to every data block in the file independently and storing the resulting hash values in the leaves of the tree. The value of every interior node in the hash tree is computed by applying the hash function to the values of its children. The value at the root of the tree, which is called the *root hash value*, then represents a unique cryptographic digest of the data in the file. Therefore, only the root hash value must be protected, and the tree data itself may be stored on untrusted storage.

A single file-data block can be verified by computing the hash value of the block in the leaf node and by recomputing all tree nodes on the path from the leaf to the root. To recompute an interior node, all sibling nodes must be read from storage. The analogous procedure works for updates. Using hash trees, the cost of a read or a write operation of integrity-protected files is logarithmic in the length of the file (in the worst case), instead of proportional to the entire file length.

The question where to store the hash-tree data must be addressed. Conceptually, the hash tree is part of the meta-data, as it contains information *about* the file data. But as the hash tree must be updated along with every data operation, its size is proportional to the size of the file, and it does not have to be protected since it resembles file data. This suggests that it should be stored together with the file data.

In SAN.FS, for example, where a file-data block is 4 kB, and using the SHA-256 hash function, a hash tree takes about 1% of the size of the corresponding file, for all but the smallest files.

Moreover, the SAN.FS protocol between the clients and the MDS is optimized for small messages that typically are of constant size. The protocol would require major modifications to handle the data traffic and the storage space needed by hash trees.

Therefore, we store hash-tree data on the untrusted storage space and only save the root hash value on the MDS together with the meta-data. We allocate a separate file object per file for storing hash-tree data. The existing functions for acquiring and accessing storage space can therefore be exploited for storing the hash tree. The file is visible at the object layer, but filtered out from the normal file system view of the clients.

4 Implementation

We have implemented a prototype of the cryptographic SAN file system design in Linux. This section describes the extensions of the SAN.FS protocol, the modifications to the MDS, and the implementation of encryption and integrity protection in the client file system driver. The storage devices have not been modified.

4.1 SAN.FS Protocol

The clients communicate with the MDS using the SAN.FS protocol version 2.1 [17]. The SAN.FS protocol implements reliable message delivery and defines requests and transactions. Either participant can send request messages to the other participant with commands that can be executed quickly. Transactions consisting of four messages are only initiated by the client and only for executing operations that result in state changes on the server.

The SAN.FS protocol defines multiple types of locks that can be acquired by clients on file and directory objects. The *data locks* on files are relevant for the cryptographic operations. A data lock protects meta-data and file data cached locally by a client from concurrent access by other clients.

A data lock on a file object is typically held in either *shared read* or *exclusive* mode. It applies to the entire file and allows the client to read or to modify file data, respectively. When the server grants a data lock on a file object to a client, it sends along the object attributes, such as file size and access permissions.

A set of *cryptographic attributes* has been added to the object attributes. The cryptographic attributes contain the type of cryptographic protection applied (encryption, integrity, or both), the encryption method, the encryption key, the hash method, the root hash value, and the identifier of the hash-tree file object. As the object attributes are always passed to the client with a granted data lock, the client knows all necessary information to perform the cryptographic operations.

The most important extensions in the protocol occur for creating a file and for accessing the object attributes.

Creating a file object: When the client sends a request to create a file object, it can also specify the desired cryptographic attributes. These flags take effect for the newly created file unless the server is configured to override them. The root hash value is left empty at this time.

Accessing file object attributes: When a client requests the acquisition of a data lock to access a file, it also receives the cryptographic attributes as part of the response from the MDS. A client holding an exclusive data lock on a file object is also allowed to modify the cryptographic attributes, for example to turn on encryption. Usually the client modifies only the root hash value in accordance

with the data that it writes. When the client returns an exclusive data lock to the MDS, the root hash value has to be consistent with the hash tree and the data in the file.

Apart from extending the SAN.FS protocol to handle the cryptographic data, the protocol traffic must be cryptographically protected on the network. This can either be achieved by establishing an IPsec tunnel between the client and the MDS or by using Kerberos to encrypt the messages between the client and the MDS. Both forms have been implemented. Using IPsec is easy because it can be configured at the operating system level. To use Kerberos, a small number of changes to the client driver and the MDS implementation have been necessary.

4.2 Meta-Data Server

Only minimal changes to the MDS are required. The cryptographic attributes are stored together with existing attributes of every file object. In contrast to most operations, where the MDS merely responds to client requests, the MDS takes an active role in setting the cryptographic attributes: It can be configured to enforce that the encryption and integrity protection flags be turned on or off, and to mandate the choice of particular encryption and hash methods. This allows the administrator to specify a uniform policy for the cryptographic protection applied to the file system.

The MDS can also generate an encryption key upon creation of a new file. It contains a cryptographically strong pseudorandom generator for this purpose.

4.3 Client Driver

Most of the cryptographic extensions are located in the client driver, because it performs the cryptographic operations on the bulk data. The SAN.FS client driver we used is implemented as a Linux kernel module for the 2.6.6 kernel. Its structure is shown in Figure 3. It consists of two main parts:

StorageTank file system driver (STFS): The STFS module contains the platform-dependent layer of the driver and implements the interface to the VFS layer of the Linux kernel. It handles reading and writing of file data from and to the page cache and the block devices.

Client state manager (CSM): The CSM is the part of the driver that interacts with the MDS using the SAN.FS protocol. It maintains the object attributes, including the cryptographic attributes. The CSM code is platform-independent and portable across all SAN.FS client driver implementations. It uses a generic interface for platform-dependent services of the operating system (not shown in the figure). Note that the CSM is not involved in reading or writing file object data.

As for any other block-device-based file system, the cached file data is maintained by the Linux page cache. All cryptographic operations operate on blocks of 4 kB at a time, which is the smallest unit of data allocation in the SAN.FS protocol. Conveniently, the page size in Linux is also 4 kB, so that the cryptographic operations do not have to span multiple paging operations.

Our cryptographic operations take place at the bottom of the client driver on the data path, immediately above the block-device layer. Read and write requests from the file system result in paging requests that are processed asynchronously by a pager module implemented in the client driver. The pager module consists of multiple threads for sending requests to read data from storage and for writing dirty pages back to disk (SAN.FS does not use Linux' `pdflush` daemon).

More concretely, when a pageout thread writes out a page of an encrypted and integrity-protected file, the thread first encrypts the page and hashes the resulting ciphertext to obtain the leaf value for the hash tree. It stores the ciphertext in a buffer page that must be allocated for the request. Then it dispatches a write request from the buffer page to the block device, according to the data layout.

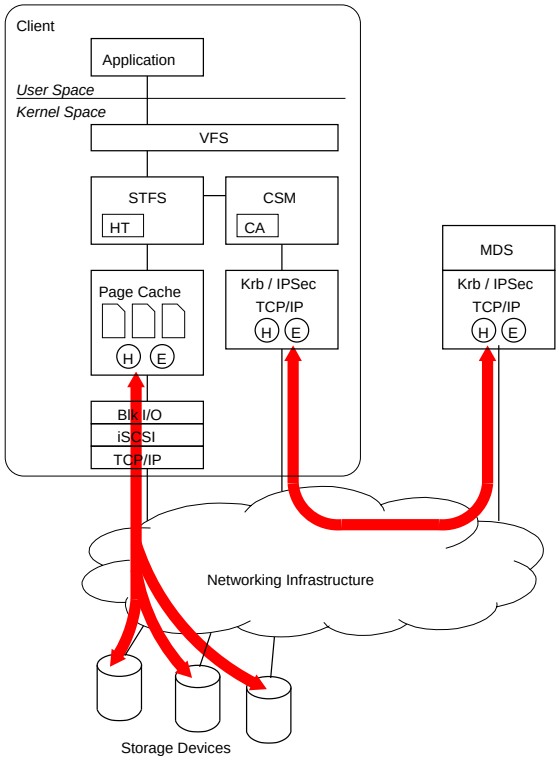


Figure 3: SAN.FS Client Implementation Overview. The encircled “E” and “H” denote encryption and hashing operations.

For pagein requests, integrity verification and decryption take place analogously in a pagein kernel thread (in kernel-thread context), after the block device has completed the I/O request and the page has been brought in completely (in interrupt context).

The other modifications concern the CSM and its data structures, through which the link to the MDS storing the hash information is established. The extension mainly deals with processing the cryptographic attributes (CA).

The driver uses the cryptographic functions in the Linux kernel crypto API for encrypting and for hashing data. This approach enables the use of a wide range of cryptographic algorithms and dedicated hardware accelerators supporting this interface.

Implementing encryption and decryption is straightforward, but the hash-tree operations require some sophisticated algorithms. The hash-tree (HT) data is buffered in the page cache, and for every node in the tree, two flags are maintained that denote whether the node has passed verification and whether a node is dirty because a write operation to a page invalidated it. Using these flags, a pagein operation only needs to verify some nodes along the path to the root until it encounters a node that has already been verified. A pageout operation on a dirty page writes a new hash value into a leaf of the tree. The internal nodes of the hash tree are only recomputed after all dirty pages that it spans have been written out. When a file is processed sequentially (for reading or writing), buffering the hash tree in this way results in a constant processing overhead per page operation [8].

One complication that arises is that to verify the integrity of a page during a pagein operation, all corresponding hash-tree data must be ready before the page arrives and its hash value can be compared with the value in the leaf node. Because verification occurs in a kernel thread when the I/O operation

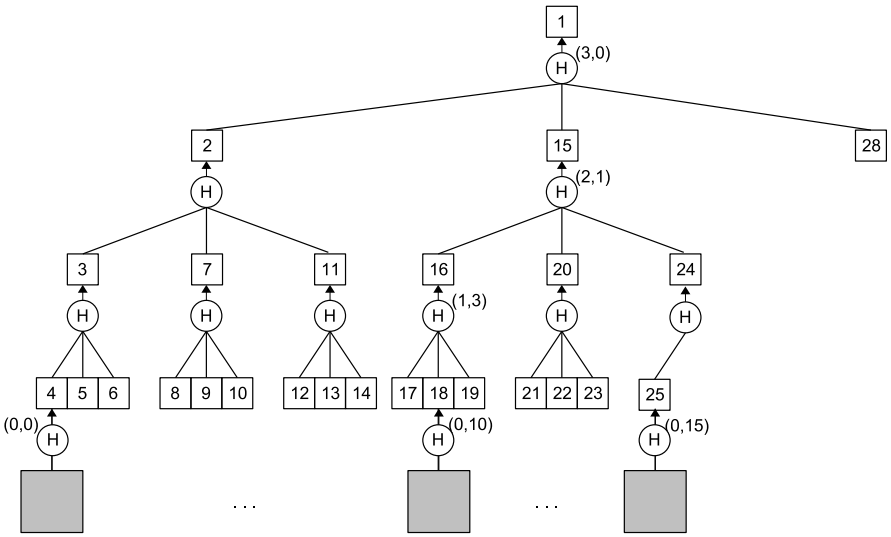


Figure 4: A ternary hash tree with four levels, numbered from 0 to 3 according to their height. The small squares represent the nodes of the tree and contain the node index in pre-order enumeration. The nodes at level 0 are the leaf nodes and are computed by hashing a single data block (grey squares). Levels 1–3 contain internal nodes, and level 3 contains the root hash.

is completed, it is not possible to start additional I/O operations for reading hash-tree data or allocating more memory in this context. Therefore, our implementation serializes the operations and ensures that all necessary hash-tree data is available before the pagein request is dispatched to the block device.

The design is also illustrated in Figure 3, where an encircled "E" stands for encryption and an encircled "H" stands for integrity protection operations. The arrows depict the flow of the protected data.

4.4 Hash Tree Layout

This section completes the description of the cryptographic SAN.FS client driver by illustrating the layout of the hash-tree data.

To compute the hash tree, a file is divided into 4 kB blocks, corresponding to the Linux page size. We recall the construction of a k-ary Merkle tree using a hash function $H()$: Every leaf node stores the output of H applied to a data page of length b bytes, and every internal node stores the hash value computed on the concatenation of the hash values in its children.

Suppose the tree has depth t. A *level* of the tree consists of the set of nodes with the same distance from the root. Levels are numbered according to their *height* in a drawing of the inverted tree as shown in Figure 4. The height of the root node is t. Every other node has height $h - 1$ if its parent has height h. Hence, leaves have height 0. The j-th node (from the left) with height h in the tree can be identified by the tuple (h, j).

As the maximum file size in SAN.FS is fixed (2^{64} bytes), the maximum depth of the hash tree can be computed in advance, given the degree k. A high degree k results in a flat tree structure and has therefore similar unfavorable properties as using a single hash value for the whole file. If k is small, the tree is deeper and therefore requires more space as well as more integrity operations during verification, especially with random-access workloads. Therefore, we chose $k = 16$ and obtain a tree of depth 13 in our implementation. The complete tree with maximum depth is constructed implicitly, but every level

contains only as many allocated nodes as are needed to represent the allocated blocks of the file. This choice simplifies the design of the hash-tree algorithms, in particular with respect to padding and file holes.

In particular, no leaf nodes are allocated for data blocks beyond the length of the file or for data blocks in holes. As reading such blocks would return the all-zero string according to the file-system semantics, we treat them as all-zero blocks for computing the hash tree. To prevent the allocation of hash-tree nodes covering empty file areas, the same heuristic encoding scheme is used for hash-tree nodes by the implementation: When a hash node is read from the hash-tree file object and returns the all-zero string, it is interpreted as the hash value resulting at that particular height of the tree when file data of only zeroes is hashed. This ensures that all leave nodes in the subtree rooted at this node contain only the hash of a block of zeroes and need not be allocated either. The node values for all-zero file data can be precomputed for all levels in the driver.

To serialize the hash tree, several choices are available: for example, level-by-level enumeration with two-dimensional identifiers of the form (h, j) or enumeration according to a recursive tree-traversal algorithm. Because of our choice to always implicitly maintain the hash tree for the maximum file size, enumerating the nodes according to a pre-order tree traversal is advantageous.

Figure 4 shows the typical case of contiguous file data starting at offset 0 using a ternary hash three with four levels. As can be verified easily, all hash-tree nodes that have to be allocated are also in a contiguous region in pre-order enumeration, starting with the root node at index 1. Using the heuristic encoding above, no unnecessary tree nodes have to be allocated for such files; all nodes that are to the left of the path from the highest leaf node to the root node correspond to the hash value of the all-zero file data, which are not allocated.

The nodes of the hash tree are serialized by traversing the tree in pre-order and writing every node to the file in that sequence. Some simple algorithms can be used to calculate the index of a node in pre-order enumeration from their two-dimensional identifier.

5 Performance Analysis

In this section, we first analyze the performance of the encryption part and then extend our benchmarks to the integrity-protection implementation. We employ two macro-benchmarks, dbench [30] and Postmark [19], representing realistic workloads, and synthetic micro-benchmarks, which illustrate specific aspects of our approach.

Our testbed consists of two storage servers (one for the meta data and one for the data to be stored), an MDS, and a client. Table 1 shows their detailed hardware and software configurations.

The meta-data storage server contains a single drive. The data storage server contains 14 drives, organized in two RAID 5EE arrays with seven drives each, in an IBM storage expansion EXP-400 using the IBM ServeRAID 6m RAID controller. All disks are IBM Ultra320 SCSI disks with 73.4 GB capacity and running at 10k RPM. The storage devices are connected with iSCSI to the MDS and the test client. The iSCSI connections use the iSCSI Enterprise target 0.4.11 and the Linux iSCSI initiator 4.0.1.7. All

Function	**Machine Type**	**CPUs**	**Memory**	**Linux kernel**
Meta-data server	IBM x335	2 hyper-threaded Intel Xeon 2.8 GHz	2 GB RAM	2.4.21-smp
Meta-data storage	IBM x336	2 hyper-threaded Intel Xeon 3.6 GHz	1 GB RAM	2.6.11.11-smp
Storage server	IBM x345	2 hyper-threaded Intel Xeon 3.2 GHz	4 GB RAM	2.4.20-smp
Client	IBM x346	2 hyper-threaded Intel Xeon 3.2 GHz	3 GB RAM	2.6.6-smp

Table 1: Software and hardware configurations of the test environment.

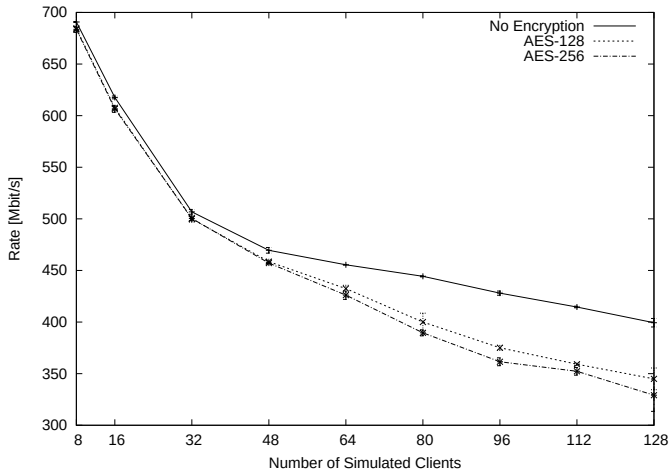


Figure 5: Encryption performance of the client driver measured with dbench; no data is actually stored on disk.

devices are connected by a single switched gigabit Ethernet, i.e., the network topology corresponds to the one in Figure 2.

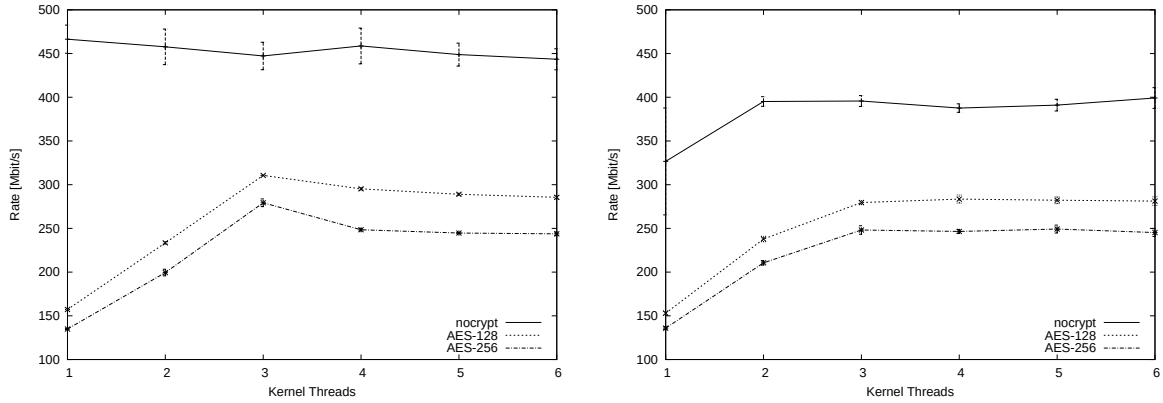
5.1 Confidentiality Protection

We report results of three different types of confidentiality-protection tests. First we present an analysis of the client driver performance, then we quantify the benefits of parallel processing in the kernel by using sequential reads and writes of a huge amount of data, and finally we examine a realistic file-system workload using Postmark.

In the first test we use the dbench file system benchmark tool (v3.04), which simulates the file system load of a Samba server and evaluates the throughput as a function of the number of clients accessing the server. In dbench, about 90% of the I/O operations are writes. The goal of this test is to evaluate the performance of the client driver, i.e., the overhead due to encryption. For this we utilize the nullio-mode on the iSCSI storage device, which means that the device immediately answers any request without actually reading from or writing to the disks. In the client, there are four pagein and four pageout threads to take advantage of all processors in the system. Figure 5 shows the results for no encryption, AES-128, and AES-256. Each curve represents the average of 5 test runs with a 95% confidence interval.

As long as the number of clients is low, most of the operations are handled by the page cache, and almost no difference between using encryption or not can be observed. With more than 48 clients, the fact that data is written to the storage device and hence encryption as well as decryption are done during this process becomes noticeable. The performance impact is fairly stable at roughly 13% for AES-128 and 17% for AES-256.

The next test consists of reading and writing large amount of sequential data using the Unix convert and copy command `dd`. Eight files of size 1 GB each are written and read concurrently in blocks of 4 kB. The eight files are organized into two groups of four, and each group is stored on one of the RAID arrays, to avoid the disks being the performance bottleneck. The goal is to keep the file system overhead minimal in order to measure the actual end-to-end read/write performance. We vary the number of kernel threads for pageout and pagein operations, which allows us to quantify the benefits of parallel processing.



(a) Read performance.

(b) Write performance.

Figure 6: Encryption performance for reading and writing large amounts of sequential data, as a function of the number of concurrent pager threads in the kernel.

The read and write rates displayed in Figure 6 are calculated from the average execution time of the eight `dd` commands, which was measured using the Unix `time` command. Encryption benefits from exploiting all four processors available in the system, but the graph also shows that it does not make sense to use more threads than physical processors. In general, when using three or more threads, the measured overhead for encryption ranges from 28 to 44%. If there is no encryption, the storage devices limit the performance to 400 Mbit/s write rate and 450 Mbit/s read rate, whereas with encryption, client processing power is the main bottleneck. Additional tests revealed that the performance using iSCSI nullio-mode achieves about 800 Mbit/s for reading and about 720 Mbit/s for writing of unencrypted data, thus saturating the gigabit Ethernet (including the TCP/IP and iSCSI overhead). For encrypted data with nullio-mode, the results remain roughly the same.

A realistic benchmark for file-system applications is Postmark, which creates a file-system load similar to an Internet mail, web, or news server. It creates a large number of small sequential transactions.

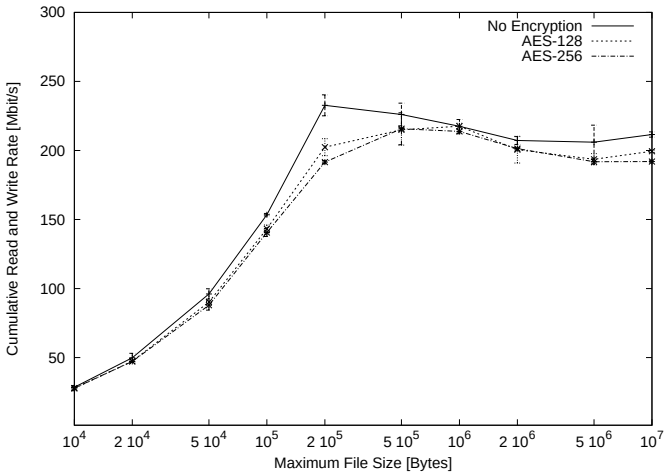


Figure 7: Encryption performance using Postmark, with varying maximum file sizes.

	No Integrity [Mbit/s]	With Integrity [Mbit/s]	Difference [%]
Read	482	303	-36.19
Write	388	384	-3.66

Table 2: Integrity-protection performance for reading and writing large amounts of sequential data.

	No Integrity		With Integrity		Difference
	[MBit/s]	[%]	[MBit/s]	[%]	[%]
No encryption	219		156		-28.7
AES-128	202	-8.0	147	-5.8	-27.1
AES-256	198	-9.6	141	-9.7	-28.8

Table 3: Integrity-protection performance using Postmark, showing the cumulative read and write rate. The "no-integrity" column shows the throughput without integrity protection, for no encryption, for AES-128 encryption, and for AES-256 encryption. The second column denotes the relative performance loss due to using encryption. Analogously, the column "with integrity" shows the same data with integrity protection applied. The "integrity loss" column denotes the relative performance loss due to applying integrity protection in each of the three cases (no encryption, AES-128, and AES-256).

Its single-threaded nature is not artificially limiting performance here as due to the networking and storage device latency, read and write requests still can be parallelized in the kernel. The iSCSI target uses the fileio-mode and writes the data to disk. Figure 7 shows the cumulative read and write rate reported by Postmark v1.51, as a function of the maximal file size parameter. The minimum file size is being fixed to 1 kB and the maximum file size varies from 10 kB to 10 MB. In this test, Postmark is configured to create 2000 files with sizes equally distributed between the minimum and maximum configured file size and executes 5000 transactions on them. All other parameters are set to their default values in Postmark. Each curve represents the average of 11 differently seeded test runs. The 95% confidence interval is also shown.

It is clear that the smaller the files are, the larger is the fraction of meta-data operations. Up to a maximum file size of 20 kB, the performance is limited by the large number of meta-data operations. Above this size, we reach the limitations of the storage devices. In general we can see that the confidentiality protection overhead is almost negligible and does not exceed 10% in this benchmark.

5.2 Integrity Protection

We measured our integrity-protection implementation with two of the benchmarks described in the previous section: using `dd` for reading and writing large amounts of sequential data and with the Postmark benchmark.

For the first test, Table 2 compares read and write rates with no integrity protection and with SHA-256 for integrity protection. The test uses the setup as described in Section 5.1, with four pageout kernel threads for writing and and four pagein kernel threads for reading. Without integrity protection the results correspond to the ones shown in Figure 6. Writing incurs no significant overhead as the hash tree is calculated and written to disk only at the end, after all file data has been written. In contrast, the read operations are slower, because the hash tree data has to be prefetched and the process may also result in a pseudo-random access pattern on the hash-tree file.

We also ran the Postmark benchmark as described in Section 5.1 with integrity protection using

SHA-256 added. Table 3 shows the reported throughput in terms of a cumulative read and write rate for a maximum file size of 20 MB and a total number of 1000 data files. The base case, with no integrity protection, corresponds to the results reported in Figure 7.

The results show that encryption has a smaller impact on performance than integrity protection. This is actually not surprising because integrity protection involves much more complexity. Recall that our implementation ensures that all hash-tree nodes necessary to verify a data page must be available before the read operation for the data page is issued. This ensures that the completion of the page-read operation does not block because of missing data. Executing these two steps sequentially simplifies implementation but introduces a delay. Furthermore, managing the cached hash tree in memory takes some time as well.

6 Conclusion

We have presented a security architecture for cryptographic distributed file systems and its implementation in IBM SAN.FS. By protecting data on the clients before storing it on a SAN, no additional cryptography operations are necessary to secure the data in-flight on the SAN. Moreover, no additional computations by storage devices and no changes to the storage devices are required. The architecture can also be integrated with future storage devices that support access control, like object storage [2].

The implementation in SAN.FS as a monolithic cryptographic file system shows that sustained high performance can be achieved. By carefully integrating the cryptographic operations in the appropriate places of the file system driver, the overhead is actually almost not noticeable in a typical file-server environment. This is consistent with earlier benchmarks of cryptographic file systems [32].

Our approach has three distinct advantages over previous systems. First, by centralizing the key management on an on-line trusted server (the MDS in our case), we gain efficiency because key management can be done with symmetric cryptography. In contrast, key management schemes performed entirely by the users, as in SFS [23] or in Windows EFS [28], requires the use of slower public-key cryptography.

Secondly, we believe that cryptographic integrity protection is an important requirement, even though many users of secure file systems first concentrate on encryption. Since integrity protection is also considerably more complex than encryption alone, most cryptographic file systems available today do not support it. Some systems, like SiRiUS [10], always hash entire files, and will not perform well with large files.

And, last but not least, many past designs of cryptographic file systems have chosen to simplify the implementation by using the layered approach. This limits their performance because they must maintain several data buffers. Some, like Cepheus [7], process data in user space, which involves copying the data in and out of the kernel multiple times. Although building the cryptographic operations into the kernel requires more work, our results show that it performs well.

Still, there is room for improvement in our design and implementation. Our hash tree implementation should include more sophisticated locking mechanisms in order to be able to read hash-tree data and file data in parallel instead of sequentially. Furthermore, the latest Linux kernel crypto API allows operations to be performed without setting up scatter lists. This would accelerate computations in the hash tree, where the use of scatter lists only adds overhead. Last but not least our choice of a 16-ary hash tree was somewhat arbitrary. In ongoing work, we are exploring different hash tree topologies and alternative ways to store the hash tree. Preliminary results show that these two factors impact the file system performance.

References

[1] 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 *Proceedings of 1st IEEE Security in Storage Workshop (SISW)*, pp. 10–23, 2002.

[2] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi, "Towards an object store," in *Proceedings of 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[3] M. Blaze, "A cryptographic file system for Unix," in *Proceedings of First ACM Conference on Computer and Communications Security*, Nov. 1993.

[4] R. C. Burns, R. M. Rees, L. J. Stockmeyer, and D. D. E. Long, "Scalable session locking for a distributed file system," *Cluster Computing*, vol. 4, pp. 295–306, Oct. 2001.

[5] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiono, "The design and implementation of a transparent cryptographic filesystem for UNIX," in *Proceedings of USENIX Annual Technical Conference: FREENIX Track*, pp. 199–212, June 2001.

[6] "Secure Hash Standard." Federal Information Processing Standards (FIPS) Publication 180-2, Feb. 2004.

[7] K. E. Fu, "Group sharing and random access in cryptographic file systems," master thesis, Massachusetts Institute of Technology, June 1998.

[8] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proceedings of 9th Int. Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[9] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, and D. Rochberg, "A case for network-attached secure disks," Tech. Rep. CMU-CS-96-142, School of Computer Science, Carnegie Mellon University, 1996.

[10] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proceedings of 10th Network and Distributed System Security Symposium (NDSS)*, pp. 131–145, Feb. 2003.

[11] V. Gough, "EncFS: Encrypted file system." `http://arg0.net/wiki/encfs`, July 2003.

[12] M. A. Halcrow, "eCryptfs: An enterprise-class encrypted filesystem for Linux," in *Proceedings of the Linux Symposium*, pp. 201–218, July 2005.

[13] M. A. Halcrow *et al.*, "eCryptfs: An enterprise-class cryptographic filesystem for Linux." `http://ecryptfs.sourceforge.net/`, 2005.

[14] L. G. Harbaugh, "Encryption appliances reviewed," *Storage Magazine*, Jan. 2006.

[15] D. Hildebrand and P. Honeyman, "Exporting storage systems in a scalable manner with pNFS," in *Proceedings of 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, Apr. 2005.

[16] R. Hölzer, "Cryptoloop HOWTO." `http://www.tldp.org/HOWTO/Cryptoloop-HOWTO/`, Jan. 2004.

[17] "IBM TotalStorage SAN File System Draft Protocol Specification 2.1." Available from `http://www.ibm.com/servers/storage/software/virtualization/sfs/`, Sept. 2004.

[18] IEEE P1619, "Draft standard architecture for encrypted shared storage media." available from `http://www.sisw.org`, Mar. 2006.

[19] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, 1997.

[20] V. Kher and Y. Kim, "Securing distributed storage: Challenges, techniques, and systems," in *Proceedings of the Workshop on Storage Security and Survivability (StorageSS)*, 2005.

[21] J. Li, M. Krohn, D. Mazires, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 121–136, 2004.

[22] D. Mazières, "A toolkit for user-level file systems," in *Proceedings of USENIX Annual Technical Conference*, June 2001.

[23] D. Mazières *et al.*, "Self-certifying file system." `http://www.fs.net/`, 2003.

[24] D. Mazières, M. Kaminsky, F. Kaashoek, and E. Witchel, "Separating key management from file system security," in *Proceedings of the ACM Symposium on Operating System Principles (SOSP '99)*, 1999.

[25] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank — a heterogeneous scalable SAN file system," *IBM Systems Journal*, vol. 42, no. 2, pp. 250–267, 2003.

[26] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology: CRYPTO '87* (C. Pomerance, ed.), vol. 293 of *Lecture Notes in Computer Science*, Springer, 1988.

[27] M. Rajagopal, E. G. Rodriguez, and R. Weber, "Fibre channel over TCP/IP (FCIP)," RFC 3821, Internet Engineering Task Force, July 2004.

[28] M. Russinovich, "Inside encrypting file system," *Windows & .NET magazine*, June–July 1999.

[29] J. Satran, K. Meth, C. Sapuntzakis, and M. C. E. Zeidner, "Internet small computer systems interface (iSCSI)," RFC 3720, Internet Engineering Task Force, Apr. 2004.

[30] A. Tridgell, "dbench v3.04." `http://samba.org/ftp/tridge/dbench/`, 2004.

[31] C. P. Wright, M. Martino, and E. Zadok, "NCryptfs: A secure and convenient cryptographic file system," in *Proceedings of the Annual USENIX Technical Conference*, pp. 197–210, June 2003.

[32] C. P. Wright, J. Dave, and E. Zadok, "Cryptographic file systems performance: What you don't know can hurt you," in *Proceedings of 2nd IEEE Security in Storage Workshop*, pp. 47–61, Oct. 2003.

[33] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright, "On incremental file system development," *ACM Transactions on Storage*, vol. 2, pp. 161–196, May 2006.

[34] E. Zadok and J. Nieh, "FiST: A language for stackable file systems," in *Proceedings of USENIX Annual Technical Conference*, June 2000.