Cryptographic Security for a High-Performance Distributed File System

Roman Pletka *
AdNovum Informatik AG
CH-8005 Zürich, Switerland
rap@adnovum.ch

Christian Cachin
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
cca@zurich.ibm.com

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 as well as 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 and iSCSI, 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, dynamic security mechanisms are required for protecting stored data in virtualized and networked storage systems.

A secure storage system should protect the confidentiality and the integrity of the stored data. In distributed storage systems, one can distinguish between *data in flight*, i.e., data in transit on a network between clients, servers, and storage devices, and *data at rest*, i.e., data residing on a storage device. Data at rest is generally considered to be at higher risk than data in flight, because an attacker has more time and opportunities for unauthorized access. Moreover, new regulations such as Sarbanes-Oxley, HIPAA, and Basel II also dictate the use of encryption for data at rest.

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, or by a virtual device driver in the operating system (e.g., encryption using the loopback device in Linux). Its advantage is that file systems can use the encrypted devices without modifications, but such file systems cannot extend the cryptographic security to its users, on the other hand. This is because 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 in Section 2, we outline the design of our implementation in SAN.FS, file system for SANs from IBM [11], in Section 3. Our design addresses confidentiality protection by data encryption and integrity protection by means of hash trees. We have implemented our design in SAN.FS and report about its performance in Section 4. The model itself as well as our design choices are generic and can be applied to other distributed file systems. Due to space limitations, we refer the reader to the full version of the paper [13] for a description of the implementation.

2. Model and Related Work

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

*Work done at IBM Zurich Research Laboratory.

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: NASA Goddard Space Flight. Downloaded on July 13,2010 at 20:46:07 UTC from IEEE Xplore. Restrictions apply.

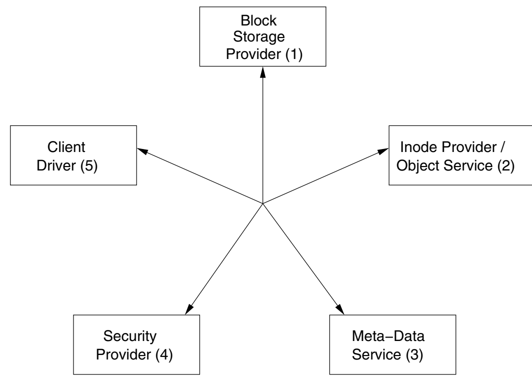

Figure 1. Components of a distributed file system.

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, as in AFS or NFSv4.

The NASD architecture [5] and its successor Object Store [1] 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 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).

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, such as hashes of all files.

File systems with enhanced capabilities such as cryptographic protection exist in two forms: either as a *monolithic* solution, realized within an existing physical file system that uses an underlying block-storage provider, or as *stackable* or *layered* virtual file system, which is mounted over another (physical) file system.

Previous Work. 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.* [16] and by Kher and Kim [9] 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. A prominent example is CFS [2], which uses an NFS loopback server in user space and provides per-directory keys that are derived from passphrases. SFS [10] is a distributed cryptographic file system also using the NFS inter-

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: NASA Goddard Space Flight. Downloaded on July 13,2010 at 20:46:07 UTC from IEEE Xplore. Restrictions apply.

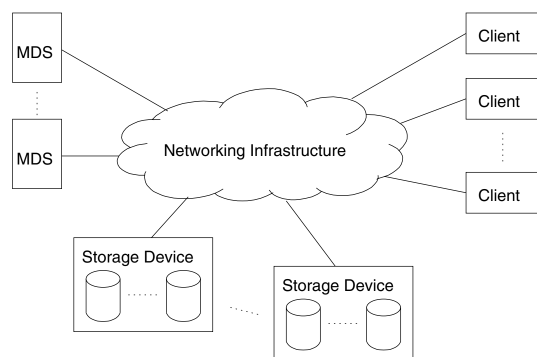

Figure 2. The architecture of SAN.FS.

faces, which is available for several Unix variants. These systems do not contain an explicit security provider responsible for key management, and delegate much of that work to the user.

SFS-RO [3] and Chefs [4] are two systems protecting file integrity using hash trees designed for read-only data distribution, where update are only possible by using off-line operations.

Microsoft Windows 2000 and later editions contain an extension of NTFS called EFS [14], 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.

Many recent cryptographic file systems follow the layered approach: NCryptfs [15] and eCryptFS [7] are native Linux file systems, which are implemented in the kernel and use stacking at the VFS layer. EncFS [6] for Linux is implemented in user-space relying on Linux's file system in user space module (FUSE).

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 offer high performance and allow file sharing and concurrent access to encrypted files.

3. Design

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 [11]. 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: 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 the model of Section 2) as an intermediate layer. Second, there is a 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. The storage devices, which are standard SAN-attached storage servers that implement a block-storage service, are the third type of components. 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 clients 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*. 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. As the clients heavily rely on local data caching to boost performance, the MDS essentially implements a cache controller for the distributed data caches at 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. 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.

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 such that 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 does not expose them to 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, 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

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: NASA Goddard Space Flight. Downloaded on July 13,2010 at 20:46:07 UTC from IEEE Xplore. Restrictions apply.

the MDS as extensions of the file meta-data. The links between clients and the MDS are protected using 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.

We remark that an alternative type of storage device, providing strong access control to the data, is available with object storage [1]. 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.

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.

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 changes to the MDS for adding it are small. The key can be chosen by either the MDS or the client.

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 functions, 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 value is computed using a *hash tree* proposed by Merkle [12]. 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.

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.

We store the hash-tree data on the untrusted storage devices 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. The SAN.FS distributed locking protocol is modified such that the hash tree object is tied to the corresponding data file and always covered by the locks for the data file. A detailed description of our implementation can be found in the full version of the paper [13].

4. Performance Analysis

In this section, we report on benchmarks of a prototype implementation of the above design, providing encryption and integrity protection. Here we give only results for Postmark [8], a benchmark creating realistic workloads, and for a synthetic benchmark, which reads and writes in parallel large amounts of sequential data. More details can be found in the full paper [13].

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. All machines are IBM x335/6 and x345/6 systems with 2 hyper-threaded Intel Xeon CPUs each and clock speeds from 2.8–3.2 GHz. The client has 3 GB RAM. 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 over a single switched Gigabit-Ethernet.

Confidentiality Protection. Postmark is a benchmark for file-system applications and generates a file-system load similar to an Internet mail, web, or news server. It creates a large number of small sequential transactions. The read and write operations generated by the transactions are parallelized by the kernel. Figure 3 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

0-7695-3025-7/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: NASA Goddard Space Flight. Downloaded on July 13,2010 at 20:46:07 UTC from IEEE Xplore. Restrictions apply.

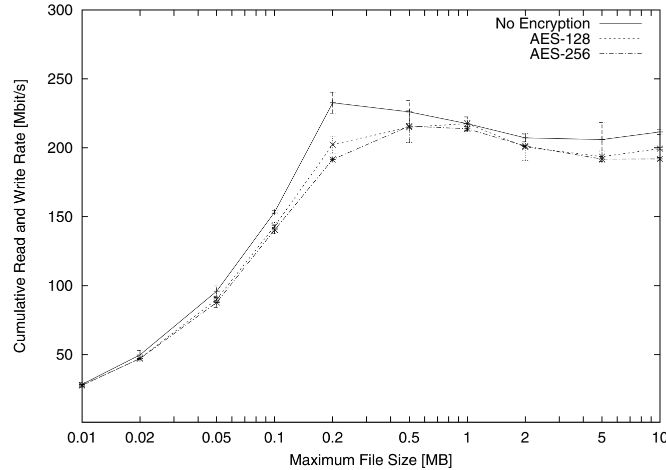

Figure 3. Encryption performance using Postmark, with varying maximum file sizes.

	Unprotected [Mbit/s]	Encrypted (AES-128) [Mbit/s]	Encrypted (AES-256) [Mbit/s]	Integrity-protected [Mbit/s]
Read	458	310	279	303
Write	388	283	247	384

Table 1. Performance comparison for reading and writing large amounts of data sequentially.

5000 transactions on them. All other parameters are set to their default values. Each curve represents the average of 11 differently seeded test runs. The 95% confidence interval is also shown, and is mostly centered closely around the mean.

The smaller the files are, the larger is the fraction of meta-data operations. Up to a maximum file size of 200 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 overhead for confidentiality protection is small in this benchmark and lies in the range of 5%–20%.

A second test consists of reading and writing a large amount of sequential data using the Unix `dd` command. 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. The implementation exploits all four CPUs visible in Linux for cryptographic operations.

The read and write rates for AES-128 and AES-256 encryption are displayed in the second and third columns of Table 1. They are calculated from the average execution time of the eight `dd` commands, which was measured using the Unix `time` command. It is evident that for such large amounts of data, the available CPU power and CPU-to-memory bandwidth become a bottleneck for performing cryptographic operations. During reads the storage bandwidth is reduced by 32% for AES-128 and 39% for AES-256, compared to not using encryption; during writes, the reduction is about 27% for AES-128 and 36% for AES-256, respectively. The measurement, however, represents an artificial worst case for a file system. Additional tests revealed that the performance using iSCSI nullio-mode, where no data is stored on disk, 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).

Integrity Protection. We describe measurements with the same two benchmarks as for encryption. We ran Postmark and applied integrity protection using SHA-256. The third column of Table 2 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 "unprotected" case corresponds to the results reported in Figure 3. The table also shows the performance of encryption and integrity protection combined.

For the other test involving large sequential reads and writes, the third column of Table 1 contains the rates with SHA-256 for integrity protection. The test uses the same setup as above. Writing shows no significant overhead because the hash tree is calculated and written to disk only after all file data has been written when dirty buffers are

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: NASA Goddard Space Flight. Downloaded on July 13,2010 at 20:46:07 UTC from IEEE Xplore. Restrictions apply.

	Unprotected		Integrity-protected		Difference
	[MBit/s]	[%]	[MBit/s]	[%]	[%]
Unencrypted	219		156		-28.7
Encrypted with AES-128	202	-8.0	147	-5.8	-27.1
Encrypted with AES-256	198	-9.6	141	-9.7	-28.8

Table 2. Performance of integrity protection and combined encryption and integrity protection using Postmark (cumulative read and write rate). The "Unprotected" columns show the throughput without integrity protection, without encryption, with AES-128 encryption, and with AES-256 encryption. The second column denotes the relative performance loss due to using encryption. Analogously, the columns under the heading "Integrity-protected" show the rates with integrity protection applied. The fifth column "Difference" shows the relative loss due to applying integrity protection for each of the encryption choices.

cleared. The hash tree size is about 1% of the size of the file. In contrast, the read operations are slower, because the hash tree data is pre-fetched and this incurs a larger latency for many low-level page-read operations. Reading may also generate a pseudo-random read access pattern to the hash-tree file.

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 first reads all hash-tree nodes necessary to verify a data page before it issues the read operation for the data page. 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 doubles the network latency of reads. Furthermore, managing the cached hash tree in memory takes some time as well.

References

[1] 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 *Proc. IEEE/NASA Conference on Mass Storage Systems and Technologies (MSST 2003)*, pp. 165–177, 2003.

[2] M. Blaze, "A cryptographic file system for Unix," in *Proc. 1st ACM Conference on Computer and Communications Security*, Nov. 1993.

[3] K. Fu, F. Kaashoek, and D. Mazières, "Fast and secure distributed read-only file system," *ACM Transactions on Computer Systems*, vol. 20, pp. 1–24, Feb. 2002.

[4] K. E. Fu, *Integrity and Access Control in Untrusted Content Distribution Networks*. PhD thesis, EECS Dept., MIT, Sept. 2005.

[5] 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, CMU, 1996.

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

[7] M. A. Halcrow *et al.*, "eCryptfs: An enterprise-class cryptographic filesystem for Linux." `http://ecryptfs.sourceforge.net/`, 2007.

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

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

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

[11] 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.

[12] 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.

[13] R. Pletka and C. Cachin, "Cryptographic security for a high-performance distributed file system," Research Report RZ 3661, IBM Research, Sept. 2006.

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

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

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

0-7695-3025-7/07 $25.00 © 2007 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: NASA Goddard Space Flight. Downloaded on July 13,2010 at 20:46:07 UTC from IEEE Xplore. Restrictions apply.