Intra-file Security for a Distributed File System

Scott A. Banachowski, Zachary N. J. Peterson, Ethan L. Miller and Scott A. Brandt
Storage Systems Research Center
Jack Baskin School of Engineering
University of California
Santa Cruz, CA 95064
{sbanacho,zachary,elm,scott}@cs.ucsc.edu
Telephone: +1 (831) 459-2545
Fax: +1 (831) 459-4829

Abstract

Cryptographic file systems typically provide security by encrypting entire files or directories. This has the advantage of simplicity, but does not allow for fine-grained protection of data within very large files. This is not an issue in most general-purpose systems, but can be very important in scientific applications where some but not all of the output data is sensitive or classified. We present a more flexible approach that uses common cryptographic techniques to secure any arbitrary-sized region of data within a file, even if the region is logically non-contiguous. This approach, called intra-file encryption, allows mixing data of different sensitivity in a single file. This benefits users by permitting related data belonging to a single file to be kept together rather than separating data of different security needs. Supporting intra-file encryption requires additional file metadata and key management services. For file systems that store metadata and files on the same server, the management of extra metadata poses little problem beyond storage overhead. However, for high-performance network-attached file systems, the additional metadata poses greater challenges related to data placement and security. This paper describes the intra-file security encryption technique with discussion of including support for it in a distributed file system.

1 Introduction

Traditionally, file system security uses an "all-or-nothing" approach—all of a file is encrypted identically. This approach is sufficient in situations where a file must be accessed in its entirety to make sense for a user or application. However, there are many cases where a user should only have access to some of the data in a file. A large file used for scientific modeling might contain mostly unclassified information, with some sections of classified

data. Other examples include a satellite map of a region containing military zones, a specification for a vehicle with sensitive information, or a recipe with a secret ingredient. Using current techniques, users that desire different levels of security must use different files, complicating access for all users.

In this paper, we introduce *intra-file security*—a flexible approach to providing end-to-end encryption in a file system. It allows users to encrypt extents of files independently from other extents, so that a single file may contain one or more secure regions. A file system incorporating intra-file security transparently handles most operations, such as automatic decryption and key management. The result is a file system with little extra programming or runtime overhead for the added functionality. Reads are entirely managed by the file system and writes occur via two separate but nearly identical function calls for unencrypted and one for encrypted data.

Flexible end-to-end encryption technology is becoming increasingly important as systems use distributed storage architectures. High-performance computer systems deal with data sets of tremendous size; files used in scientific computing and data-mining applications commonly extend beyond the capabilities of single storage devices. Distributed storage architectures provide one solution for the demands of increased storage needs. By spreading file system data over multiple network nodes, distributed storage provides high data rates through parallelism, and large, scalable storage capacity with a capability for fault tolerance through redundancy. However, distributing storage also increases the number of potential points for network intrusion, making data susceptible to security breaches. To secure sensitive data, networked file servers should store and transmit only encrypted data, which is decoded by clients with cryptographic keys. Many end-to-end encryption tools exist, and the least cumbersome for users are those built into the file system [1]. Such file systems transparently decode encrypted data for users with proper permission rights.

Existing cryptographic file systems secure data on a per-directory [1] or per-file [4] basis. This level of granularity is not flexible enough to support applications that benefit from encrypting smaller regions within files. If information is only encrypted on a per-file basis, then a set of data containing a mix of sensitive and unclassified data must be stored in two or more files, one for each security level. However, in some cases it is beneficial to keep data in a single file; users and tools can manage the data as a single entity in the file system, and the same applications may use secure and insecure data sets. Because they encrypt whole files or file systems, existing cryptographic file system techniques cannot address this problem.

Intra-file security offers additional security by allowing more fine-grained control file access, breaking a file into regions of differing security without compromising single-file semantics. This allows the system to transparently handle security operations, making the security invisible to authorized users and thus more likely to actually be used. In order to implement intra-file security, we introduce security-related metadata, and provide a key management solution that allows flexibility in security and access policy.

Section 2 introduces the intra-file security (IFS) encryption algorithm. The algorithm, based on well-known cryptographic techniques, may be implemented stand-alone or as

part of a larger system, such as a file system. Section 3 describes how to integrate IFS into a distributed object-based file system. Sections 4 and 5 discuss some possible IFS applications and related work.

2 Intra-File Security

Intra-file security (IFS) allows encryption to be applied to segments as small as a byte or as large as an entire file; multiple encrypted segments need not be logically contiguous within the file. In an IFS file, encrypted data is stored logically in-place, and occupies the physical file blocks that would have contained the unencrypted data. To support efficient random file access, we independently encrypt data from each logical file block, so there is no dependence on information from other blocks. Consider the file shown in Figure 1, which contains a non-contiguous region that must be kept secure. The region spans one entire logical block (L_1), and two partial blocks (L_2 and L_3). As mentioned above, this region is not independently encryptable using standard techniques. With IFS, this non-contiguous region of the file can be encrypted independently and made available only to appropriate users. Furthermore, because the encrypted data is left in place, all programs written to work with the full data set (including legacy applications) can still function properly. All regions of the data, encrypted and unencrypted alike, will still be readable except that the encrypted regions will not contain the secured data values but will instead contain apparently random values.

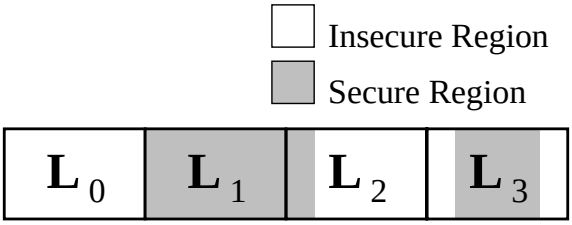


Figure 1: A single logical file address space broken into secure and insecure regions.

The encryption technique may use any block or stream cryptographic algorithm. Because the size of encrypted data in a file block may not match cipher block sizes, the algorithm is well-suited to stream ciphers, but can also be made to work with block ciphers with little additional effort. The flexibility of choosing any cryptographic algorithm allows system builders to vary encryption strength, conform with specific standards, or integrate off-the-shelf hardware chips into the system. The choice of block or stream cipher presents only a slight variation on the technique, so we present methods for both.

2.1 Block Cipher Technique

In an IFS file, secure segments may reside anywhere within a block, and may not be physically contiguous within a block. This causes a problem for block encryption algorithms

that expect to receive contiguous blocks of data for encryption. Our system combines all segments within a block into a temporary buffer before encryption, encrypts the buffer, and then redistributes the cipher back into the positions of the original plain-text segments. This process uses scatter-gather, minimizing actual copies to the bytes at the start and end of a region necessary to pad out the encryption block (often 64–128 bits), and uses pointer manipulation to do the rest of the encryption in place.

Because the output of a block algorithm is a fixed size, and the data may not necessarily match this size, we employ *cipher-text stealing* [2] to match encrypted data sizes to unencrypted sizes. Cipher-text stealing allows us to output ciphers of the same size as the input, even if they do not match the cipher block size. The encrypted data is then redistributed back to the file block in the area originally occupied by its plain-text counterpart. By using *initialization vectors* (IVs) [13] and *cipher block chaining* (CBC) [13], we also obscure data containing repeated patterns (such as headers) The IV must be unique for each block in a storage device but need not be secret.

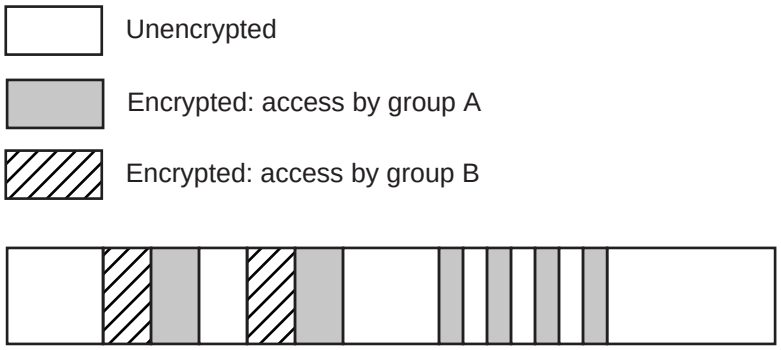
2.2 Stream Cipher Technique

By using a stream cipher such as RC4 or SEAL [13], IFS does not need to assemble data into temporary buffers or use pointer manipulation to collect bytes for encryption; instead, data may be encrypted in place. Stream ciphers such as RC4 claim a speed improvement of 10 times over DES, further improving performance. Applying feedback chaining to the stream hides data patterns—we use an IV to initialize the feedback chain, therefore the metadata structure does not differ from block mode encryption.

2.3 Encryption Metadata

By default, all data in the file is assumed to be unencrypted. In order to locate the secure data within the file, and to find the encryption parameters, each encrypted block requires a description of the location of secure segments and initialization vector information. In IFS, the structure holding this data is a security node, or *s-node*, shown in Figure 2. The size of an s-node depends on the number and layout of secure regions. A secure region is defined by an extent consisting of a start and a length; the start is relative to the start of the previous secure region, or the start of the block for the first region. Because many secure regions are formed of repeating patterns of data of varying levels of security, there is also a shorthand way of representing simple patterns of secure regions that are a fixed length and fixed distance apart. This is accomplished by specifying a repetition count associated with the offset and length specified in the secure region specification.

In addition to information about the location of secure regions, s-nodes must store the information necessary to encrypt and decrypt the secured data. This includes key information for the region as well as an initialization vector (IV)—a number used to seed the encryption algorithm when it operates on the encrypted data in the block. An IV is necessary to ensure that encrypted regions with the same data do not result in the same ciphertext, providing



Start	Length	Count	s-group
512	256	1	A
256	256	1	B
512	256	1	A
256	256	1	B
768	128	1	A
256	128	3	A

Figure 2: A 4 KB block encrypted with intra-file security and its associated security node (s-node). Note that the last entry in the s-node has a repeat count of 3, representing the three repeated secure regions near the end of the file. The first of the four regions must be represented separately because its distance from the previous region is larger than that of the following three regions.

insight about the file's structure or contents that might prove useful to an intruder. The IV must differ for each file block, and thus is a function of the logical block number as well as per-file values such as file identifier. If the IV for a block can be determined *solely* from the logical block number and per-file constants, it need not be stored in the s-node because it can be calculated at runtime.

Pointers to keys, on the other hand, must always be stored in the s-nodes. It might be possible to avoid storing key information in the s-node by simply referring to key information for the whole file; however, this approach would not permit encrypting portions of a file with different keys. Instead, we store an *s-group* identifier for each secure region; this identifier is translated by the system into a key using the approach discussed in Section 3.1.

There is one s-node structure for each logical file block that contains any encrypted segments. Note, however, that it is possible to group file system blocks together to reduce the amount of storage required by s-nodes; this technique is particularly effective for files that require large numbers of identically-sized regions with constant spacing. In such files, a few secure region descriptors can suffice for a large number of secure regions, reducing the file system overhead for IFS. Because s-nodes are allocated by the file system from the same pool of blocks used for regular files, reducing the size of security information allows more data to be stored in the file system.

It should be noted that while they are adequate for their intended purpose, the s-node structure described in this section could be improved in several ways. The s-node as depicted in Figure 2 is simple to implement, but uses space inefficiently. Instead, s-nodes could be compressed using gamma compression [14] or other techniques for compressing small numbers. Additionally, an IFS system could attempt to recognize and represent more complex encryption patterns, albeit at the cost of added complexity.

3 Integration with an OBSD File System

Although IFS may be used in any type of file system, we present a design to implement intra-file security for a file system based on Object Based Storage Devices (OBSDs). We are proposing the use of OBSDs for high-performance network-attached storage devices; this approach has similarities to Network-Attached Secure Disks (NASD) [3]. An OBSD-based file system is designed for high-performance computing workloads—precisely the kinds of applications that benefit from intra-file security. Because OBSDs require strong security in order to keep data safe in storage and transit [7], we expand the end-to-end encryption capabilities by incorporating IFS.

OBSD-based storage systems have the potential to improve both file system performance and functionality by building a high-performance storage system from inexpensive storage components connected by high-speed networks. The main hardware component of the storage system is an object-based storage device—one or more disks (or other storage devices) managed by a single CPU and seen by the file system as a single device. Data is distributed across many OBSDs, with high bandwidth coming from large numbers of concurrently operating OBSDs.

Each OBSD is responsible for managing and allocating its own storage; requests to an OBSD are of the form "write (or read) this range of bytes from file X," with low-level placement of the data and free space management left to the OBSD. High-level information such as the striping pattern across OBSDs and translation of names to file identifiers are left to a metadata server (MS), which is accessed by the user only when a file is opened or closed. This file system design is shown in Figure 3.

The key advantage of OBSDs in a high-performance environment is the ability to delegate low-level block allocation and synchronization for a given segment of data to the device on which it is stored, leaving the file system to decide only on which OBSD a particular segment should be placed. In such a distributed file system, s-nodes are stored physically near the blocks they describe, avoiding extra traffic to central servers on distributed storage systems and amortizing I/O usage among the devices. OBSDs use their own allocation policies to manage local data, including file and s-node data, placing them for efficiency within physical storage devices. Because s-nodes do not contain secrets, end-to-end encryption is provided to users without any extra involvement of the OBSD—the OBSD sends all file data and s-nodes in the clear on insecure networks. The security of encrypted data lies with the key management policy.

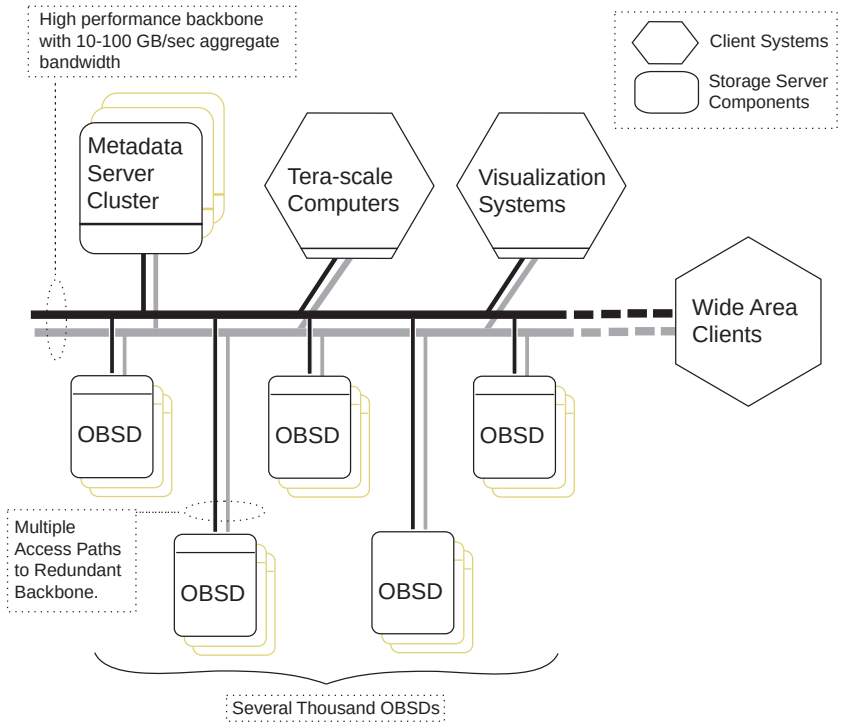


Figure 3: OBSD storage system architecture.

3.1 Authentication and Key Management

An authentication system is required for file system security, regardless of end-to-end encryption capabilities. Since we are focusing on support for intra-file encryption, a full development of the authentication system is beyond the scope of this paper. However, we rely on an authentication system for distribution of encryption keys, so we briefly describe how such a system may be implemented.

A major role of a metadata server (MS) is to control access to the file system. When users wish to open a file, the MS checks file permissions before granting access. As a first step, client software authenticates a user's identity, using standard authentication techniques such as Kerberos [9] or cryptographic hashes [7, 10]. The MS proceeds to check permission for a requested file operation using the file system's access control mechanism. However, OBSDs handle read and write requests directly; in order to enforce access rights, OBSDs must also check identities and permissions as well. The overhead of maintaining and checking access permissions at each OBSD defeats the high-performance requirement, so an OBSD uses a more efficient method to check the validity of a client's request. The MS generates *tokens* containing encoded access rights during open requests, and sends them to clients along with the file's metadata. Clients present these tokens with their requests to OBSDs. By checking the permissions encoded in the token, an OBSD determines the validity of the request. Tokens are equivalent to *capabilities* used in NASD for the same purpose [4, 3]. In IFS, security information is included in the forwarded tokens.

Access to encrypted segments is based on IFS group permissions, which we call *s-groups*. An s-group contains a list of users and/or groups that may use the key for an encrypted segment; the creator of an encrypted segment specifies s-group members during the initial write. A key server (KS) manages s-groups separately from file-access group permissions normally associated with file services; the goal is to remove management of encryption from traditional file system administration. The KS is responsible for checking s-group permissions, and generating and storing keys. From a user's viewpoint, calls to the MS involve both the MS and the KS, whether they reside on single or concurrent machines.

3.2 File I/O Interface

IFS uses standard POSIX file semantics by instrumenting interface libraries to handle security operations transparently. However, supporting encryption requires some new functions that allow writing of encrypted segments. Applications writing *only* unencrypted data and reading *any* data use the normal write and read function interfaces.

Reading encrypted data is transparent to the user. When reading data, users with a key see decrypted data when they read data; thus, applications reading data stored with IFS do not need any modifications, though they must be capable of dealing with garbage data in the data file—reads from encrypted segments of a file appear as random bits if the user lacks the proper key. If the user has the necessary key, the file system client transparently decrypts the file using keys supplied with authentication tokens. Only users with the proper key may decrypt secure segments and view the contents; the encoding of the token tells the OBSD whether or not to send s-nodes with data, so extra traffic is avoided when possible.

Under IFS, the interface to the file system is extended to support encrypted writes. Encrypted segments remain read-only unless the user has *encrypted write access*, which is granted through IFS s-group permissions. Even for users with permission, encrypted writes are explicit. Two new system calls support encrypted writes. One function translates an s-group specification into an integer identifier. The identifier is used in subsequent calls to the `secure_write` function, which is identical to the standard 'C' `write` function except for this additional argument. When writing encrypted segments, the file system client creates s-nodes for the corresponding blocks, and sends the s-nodes to the OBSD along with the blocks. When over-writing data in blocks already allocated to the file, the client must fetch and update the existing s-node (read-modify-write operation).

Unencrypted write requests to file blocks containing encryption must be carefully controlled, because users without encryption rights cannot overwrite the encrypted region of the block. To protect the integrity of encrypted data, it is impossible for users to write to encrypted segments using the traditional write function call. In order to minimize the latency of unencrypted writes, the OBSD quickly caches all data on writes, and during periods of inactivity discards changes to encrypted segments before committing the write. Essentially, this makes all encrypted segments read-only unless invoking the `secure_write` function. This policy does not impact blocks without encrypted segments, but it effects the coherency of blocks that do—until the write is fully committed, multiple copies of a block

reside in the file system. As a trade-off between performance and safety, we prefer that writes to encrypted segments do not occur unless made explicit, even for users with a key.

4 IFS Applications

To support encryption of data within existing unencrypted files that have been migrated to an IFS file system or written with non-IFS legacy applications, an IFS-capable copy program can be provided to encrypt the appropriate portions of the file. This program would take as input an unencrypted file and a specification of the regions to be encrypted.

Databases that use a single large flat-file could easily benefit from IFS by encrypting those fields of the database that must be kept secret, while still maintaining single-file semantics for the whole database. Most databases support encrypted fields by simply supplying keys for particular fields; however, this approach requires a reasonable amount of support from the database system or the database queries to remain transparent to users. By using IFS, this process could be made transparent, particularly if databases exchanged information with the file system.

Many very large files used in military and government scientific work will also benefit from IFS. Removing the need to fragment files that naturally require multiple levels of security will simplify applications as well as data management; no longer will users need to create several files with different encryption levels and keep track of which ones are related and how. Eliminating fragmentation ensures high-performance sequential and random access. Importantly, legacy applications can transparently be made IFS-capable, since the data formats and locations within the files remain unchanged even as portions of the data itself are encrypted.

IFS may also be used to transfer partial files in a distributed file system, as suggested by Muthitacharoen *et al.* [8]. By integrating IFS into a low-bandwidth distributed file system, users could gain secure access to their files even from slow clients.

5 Related Work

There have been many file systems and storage systems that provide higher security by encrypting files and metadata. Reidel, *et al.* [11] provide a good framework for evaluating secure file systems; their work discusses file systems and the security that each provides. Intra-file security is not one of their criteria; although they do discuss the granularity of key protection, the minimum protection unit is a single file.

Some file systems, such as CFS [1] and Cryptfs [15], require users to manage their own keys. This approach is simple, but is not suitable for IFS because of the sheer number of keys required [12]. Other systems such as SNAD [7], SFS and SUNDR [6, 5], and NASD [3] automatically manage encryption keys, though they do not permit partial-file encryption. Moreover, many of these systems, including NASD and SFS, store data on

the disk in an unencrypted form, using encryption only for authentication. The techniques described in this paper are based on those used in SNAD—it provides strong protection by encrypting data end-to-end, leaving it in the clear only on the client.

Intra-file security is particularly important for large, distributed file systems such as those enabled by NASD [3] and object-based storage devices (OBSDs). Reed, *et al.* provide a method for strong authentication in such an environment in SCARED [10], providing an excellent platform for both standard security [7] and the intra-file security proposed in this paper.

6 Conclusions

Secure file systems and distributed storage networks currently permit encryption only on a per-file or per-directory basis. However, there are many applications that would benefit from the ability to encrypt data in smaller pieces, using different keys to permit parts of a file to be read and written by different groups of users.

This paper presents a solution to this problem, by introducing a concept called intra-file security, and provides a high-level design for implementing it in a distributed file system and on individual servers within such a file system. Intra-file security uses additional metadata to maintain information about secure segments, allowing blocks of a file to be encrypted and decrypted individually on the client. A key management system provides group management facilities that are well adapted to the hierarchical nature of access to classified materials present in organizations requiring security.

Acknowledgments

We thank Randal Burns for his feedback and advice, and Ahmed Amer for proof-reading. We also thank our shepherd, Jack Cole, for his helpful suggestions and patience.

References

[1] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the First ACM Conference on Computer and Communication Security*, pages 9–15, Nov. 1993.

[2] J. Daeman. *Cipher and Hash Function Design*. PhD thesis, Katholieke Universiteit Leuven, Mar. 1995.

[3] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Interational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[4] H. Gobioff, G. Gibson, and D. Tygar. Security for network attached storage devices. Technical Report TR CMU-CS-97-185, Carniege Mellon, Oct. 1997.

[5] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139, Dec. 1999.

[6] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 99–104, Schloss Elmau, Germany, May 2001.

[7] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, Monterey, CA, Jan. 2002.

[8] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001.

[9] B. C. Neumann, J. G. Steiner, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Technical Conference*, pages 191–201, Dallas, TX, 1988.

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

[11] E. Reidel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, Monterey, CA, Jan. 2002.

[12] P. Reiher, T. Page, G. Popek, J. Cook, and S. Crocker. Truffles—secure file sharing with minimal system administrator intervention. In *Procedings of the 1993 World Conference on System Administration, Networking, and Security*, Apr. 1993.

[13] B. Schneier. *Applied Cryptography*. Wiley, New York, NY, 2nd edition, 1996.

[14] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 1999.

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