SNARE: A Strong Security Scheme for Network-Attached Storage

Yingwu Zhu
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221, USA
zhuy@ececs.uc.edu

Yiming Hu
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221, USA
yhu@ececs.uc.edu

Abstract

This paper presents a strong security scheme for network-attached storage (NAS) that is based on capability and uses a key distribution scheme to keep network-attached storage from performing key management. Our system uses strong cryptography to protect data from spoofing, tampering, eavesdropping and replay attacks, and it also guarantees that the data stored on the storage is copy-resistant. In spite of this level of security, our system does not impose much performance penalty. Our experimental results shows that, using a relatively inexpensive CPU in the storage device, there are little performance penalty for random disk accesses and about 9-25% performance degradation for large sequential disk accesses ($\geq$ 4 KB).

1. Introduction

Traditionally, disk drives have been bound to server machines that are supposed to be responsible for most aspects of data integrity and security. However, the demand for greater scalability has forced storage to adopt a decentralized architecture. Therefore, the network-attached storage has begun to replace traditional centralized storage systems [1, 11, 12]. In such systems, disks are attached directly to a network and achieve their scalability by eliminating single-server bottleneck.

By attaching storage directly to the network, the storage is now a first-class network citizen and exposes disk drives to direct attack from adversaries. Thus, the storage has to rely upon its own security rather than using the server's protection to defend against potential attacks.

Many secure storage systems have been proposed to achieve different levels of security. The Secure File System (SFS) [17] provides mutual authentication of servers and users using self-certifying pathnames, and NASD [11, 12] uses encryption to provide network security and authentication. However, both systems store data in clear, providing no protection of data privacy. The Cryptographic File System (CFS) [3, 4] encrypts directories and files on disks, but lacks features for sharing encrypted files among users. Even though TCFS [5] and SUNDR [18] provide encryption and authentication, they both suffer from a relatively high performance penalty. SNAD [19] uses a decentralized security architecture to provide data privacy and integrity, by pushing key objects and certificate objects into the storage.

Since network-attached storage devices are very cost-sensitive, they are resource poor relative to modern workstations and servers in terms of DRAM, computational capacity, and etc [12]. The limited resource and contention from multiple sources for memory in network-attached storage therefore motivate a design that limits the amount of memory consumed for security purpose. In this paper, we present SNARE, a strong security scheme for network-attached storage which is based on *capability*, a well-established concept for regulating access to resources [6]. In particular, our main contributions are:

- A key distribution scheme is employed to keep network-attached storage from having to perform key management.
- Our system supports user authentication and relies upon strong cryptography to protect data privacy and integrity. Using HMAC [14] instead of more performance-intensive authentication methods such as public-key encryption and performing encryption/decryption at the client minimize the effort required by the network-attached storage device's CPU.
- Our system guarantees that the data stored on the storage is *copy-resistant* [16]. That is, the data stored on a network-attached storage device could not be simply copied to another network-attached storage device with a different secret key.
- Using stored data checksums (*which are precomputed at the client*), the storage can transmit secure data

Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)
1060-9857/03 $17.00 © 2003 IEEE

faster than the client is able to verify the data, thus shifting the bottleneck from the storage to the receiving client for good scalability.

- We show that, despite this level of security, using a relatively inexpensive CPU in a network-attached storage, there are little performance penalty for random disk accesses and about 9-25% peformance degradation for large sequential disk accesses ($\geq$ 4 KB).

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the system specification. Section 4 details the cryptographic protocols. Section 5 evaluates the performance of our security system. Finally we conclude in Section 6.

2. Related Work

Much of recent storage security work has focused on authentication, data integrity, and data privacy. Many systems have been designed to tackle the security issues from different angles. Most of these systems however have security shortcomings, suffering either from weak security, poor performance, or both. Our system instead aims to provide strong security while preserving good performance on network-attached storage.

Many file systems provide access control policies to determine who can access data. NFS is one of the oldest and most widely used network file system. It offered little security until recently. The recent NFSv4 specification [23] proposes at least three security mechanisms: one using Kerberos [25] and two using a public key infrastructure. All these essentially set up a secure communication channel and enable mutual authentication. It also greatly expands the use of ACLs for access control, very similar to AFS ACLs.

Systems like AFS [13, 24] and NASD [12, 11] use Kerberos to provide security by requiring users to obtain "tickets" (or "tokens") from a third party. The tickets are then presented to the AFS file server or NASD disk as proof of identity and access rights. These two systems, however, store data in clear on disks. Thus, data privacy can not be guaranteed. Furthermore, if the system wants to prevent the leaking of data in transit (by snooping on the network), it has to perform data encryption before transmitting the data, resulting in much performance penalty. Our system instead stores data in encrypted form on disks while performing encryption at the client.

SCARED [20] provides a mechanism for authentication and protects data integrity, but does not implement end-to-end data encryption, leaving that for the underlying file system.

SNAD [19, 7] encrypts all data at the client and gives the storage sufficient information (including key objects and certificate objects) to authenticate the writer and the reader sufficient information to verify the end-to-end integrity of the data. SNAD aims to provide a decentralized security system for network-attached storage by pushing key objects and certificate objects into the storage. However it still suffers from several problems. First, the permission of the operation specified in each request to the storage must be checked by referring to the key object and certificate object. Considering the network-attached storage device having relatively small amount of DRAM (say, about 8 to 16 MB), this operation will compete for memory with the data cache and metadata cache, thus probably incurring additional expensive disk I/Os. The limited resource and contention from multiple sources for memory in network-attached storage therefore motivate a design that limits the amount of memory consumed for security function. Second, it uses public-key authentication for writing key objects. This operation is very expensive and imposes high performance penalty. SNARE instead avoids using public-key encryption on the storage. Furthermore, as the number of network-attached storage devices increases, security easily becomes a management and usability nightmare. Worse yet, the wrong key management policy harms security or severely inconveniences people [17]. So it is desirable to pull the key management out of the storage and minimize the corresponding overhead.

The SFS [17] addresses the problem of mutually authenticating the servers and users and separates key management from the file system security. Furthermore, SFS requires that users trust the storage server to store and return file data correctly. SFS-RO [9], however, does not impose such a requirement. SFS-RO is designed to support storage and retrieval of encrypted read-only data. The SUNDR file system [18] securely stores data on untrusted servers, which are managed by people who have no permission to read or write data in the file system. However, its use of digital signature will incur a relatively high performance penalty.

Many file systems are also designed to protecting data on disks. The CFS [3, 4] encrypts directories and files stored on disks using a secret key. CFS is designed as a secure local file system, therefore lacking features for sharing encrypted files among users. Furthermore, it does not protect against attacks where the bits on disks are compromised. CryptFS [29] extends CFS to be more efficient by building it as a stackable file system rather than a user level server. However, like CFS, CryptoFS still has similar sharing and authentication issues. Cepheus [8] adds file sharing to a CFS-like file system using mechanism similar to UNIX groups, while providing confidentiality and integrity of data.

TCFS [5] uses a lockbox to store a single key and encrypts only file data and file names; directory structures and other metadata are left un-encrypted. Furthermore, TCFS is relatively slow, reducing file system performance by more

than 50%.

Survivable storage is also proposed to guard against destroy attacks in collusion with storage servers. There are generally two mechanisms to prevent destroy attacks. One mechanism is to recover from the total loss of a storage server by keeping multiple copies of the data [28, 15]. Another mechanism is to protect data from unauthorized modification by using versioning on the storage servers, allowing data to be reverted to a state before intrusion [26]. However, our focus is not on the survivable storage.

Riedel et al. propose a framework for evaluating storage system security [21]. They show that encrypt-on-disk systems offer both increased security and improved performance over encrypt-on-wire. They provide us with a guidance to develop our security scheme for network-attached storage.

3. System Specification

This section specifies the infrastructure SNARE requires, keys it uses, the secure storage model it presents to file systems, and basic operation it provides.

3.1. Required Infrastructure

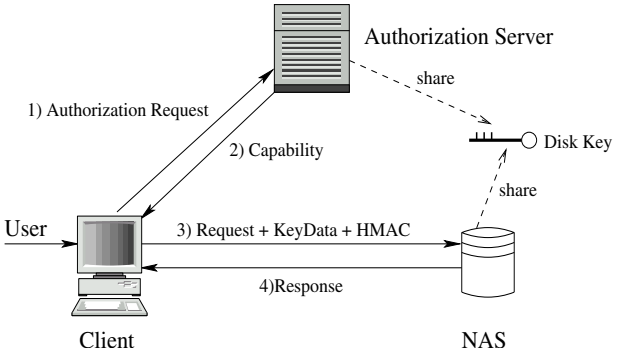

Figure 1. Interaction in SNARE

Below is the infrastructure required by SNARE, as depicted by Figure 1.

- *User*, which includes owners, readers, writers, and so on.
- *Client*: a multi-user workstation. A *client daemon* and one or more *user agent*s run on this machine. The client daemon communicates with user agents, the authorization server and the network-attached storage. The user agent acts on behalf of the user and performs encryption/decryption at the request of the client daemon. Each user has a user agent and the user agent can access its user's private key.
- *Authorization server*, which administers the name space and access control policy of the file system. It makes the access control decisions according to the policies of the high-level file system. The authorization server could be composed of a cluster of servers to avoid single point of failure and bottleneck.
- *Network-attached storage (NAS)*, which enforces the access control policy previously determined by the authorization server. It stores and retrieves files for client daemons.

3.2. Keys

Each user in SNARE has a pair of private and public keys, which will be used for user authentication. The authorization server has a pair of private and public keys. Note that SNARE uses a key distribution scheme to keep the NAS from performing key management in order to minimize corresponding overhead. So each NAS itself only contains a unique key: the disk key K_d. In addition, the disk key is shared by the authorization server.

Every file in SNARE is divided into several blocks, and each block is encrypted with a symmetric key, called a *file-data key* (i.e., per-file basis). The lockbox (as will be discussed in section 3.3.1), which refers to a key encrypted with another key, holds the file-data key for the file and is read and written by a *file-lockbox key*. File-lockbox keys are symmetric keys and are given to valid readers and writers by the authorization server. The potential benefit of using the lockbox is that it allows the authorization server to do *key aggregation*, thereby reducing the number of keys the authorization server needs to manage, distribute and receive. For example, in the context of the sharing semantics of UNIX file systems, if a set of files are owned by the same owner, the same group, and have the same permission bits, then they are authorized for access by the same sets of users. So the authorization server could use only a single file-lockbox key for these files. Note that these files each has its own file-data key. But these file-data keys do not need the authorization server to keep track of. They are transparent to both the authorization server and the NAS. As a result, using lockbox allows the authorization server to group files into logical groups so that file-lockbox keys are shared among files in each logical group *without compromising security* (due to the fact that each file has a unique file-data key), thereby reducing the number of keys the authorization server need to keep track of.

A cryptographic capability in SNARE contains two parts, namely a secret key hk_u and a secret key data $KeyData$. The secret key hk_u is derived by hashing the $KeyData$ with a K_d (where the K_d is the disk key for a particular NAS). The $KeyData$ may include user's access rights on one or more file objects. When a user ac-

cesses a file object on NAS for the first time, the user has to obtain a capability for this file object from the authorization server (as shown in Figure 1). The authorization server performs the access control policy of the file system, generates a capability by using a corresponding disk key K_d, and sends the capability back to the client. It is worth pointing out that the file-lockbox key for the file is also sent back to the client along with the capability. For each subsequent data request on this file object, the client sends the request together with the capability to the NAS bypassing the authorization server. The NAS then enforces the authorization server's exact access control decision (specified in $KeyData$) before sending a response.

In the context of secure storage, key revocation is extended so that a user's access rights to a particular piece of data can be revoked. Four ways of revoking keys [1] are provided: keys have a lifetime (controlled by $KeyData$, which can include an expiration time), valid keys are controlled by an access control version number AV on file objects, a blacklist of invalidated keys are maintained at the NAS, and all keys for a specific NAS can be revoked by changed its disk key K_d.

3.3. Secure Storage Model

3.3.1. Data Structure

Network-attached storage systems provide a richer, typed, variable-size (file object), hierarchical interface [11, 10]. Therefore, using an object interface rather than a fixed-block moves data layout management to the storage, gives the storage directly knowledge of the relationships between disk blocks, and minimizes security overhead. Since SNARE employs a key distribution scheme to keep NAS from having to perform key management, there is only one main data structure on the storage: *file object*.

Figure 2 shows the data structure of the file object on the NAS. A file object is composed of one or more encrypted data blocks along with per-file metadata. The data block is encrypted at the client with a file-data key using a symmetric encryption algorithm RC5 [2, 22]. A data block is the minimum unit of data that can be read or written in system. Note that the file-data key is a key used for encryption/decryption of data blocks, and is generated by the user (i.e., the file owner) upon the creation of the file. Rather than storing it in the clear, the NAS stores it encrypted with a file-lockbox key. Hence, the *lockbox* holds the file-data key encrypted with the file-lockbox key. It is worth pointing out that the *lockbox* is generated at the client and sent to the NAS, which is ignorant of both the file-data key and the file-lockbox key.

[1] We here mainly refer to the revoking of capabilities. The revoking of both file-lockbox keys and file-data keys can be performed lazily for performance consideration, e.g., at the time of file updates.

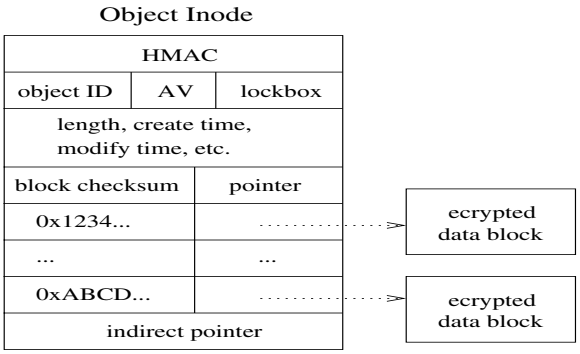

Figure 2. File object

HMAC is a keyed message authentication code, generated by hashing the file object inode information (except the *HMAC* field itself) with the disk key K_d. *Object ID* is a unique identifier for the file object on this storage device. AV is an access control version number, and it allows the authorization server to invalidate outstanding capabilities (being held by users) on an object when the access control policy for that object changes. Another copy of AV is also maintained at the authorization server.

Block checksum is computed at the client by SHA-1, stored together with their corresponding direct *pointer*s in the inode. This will always make the checksums available without additional I/O operation as long as the inode is in memory, allowing the storage to overlap the computation of the response's HMAC and retrieval of the data block for read requests. *Pointer* is stored in encrypted form, calculated by XOR encryption with a HMAC key (generated by hashing *object ID* using the disk key K_d). In addition, the inode does not maintain an *access time*, thereby avoiding updating the *HMAC* upon every request on this object.

3.3.2. Data Security

Data security in SNARE is defined by four attributes: user authentication, data privacy, data integrity and copy resistance.

User authentication is used to restrict who may or may not have access to network-attached storage. In SNARE, a user first has to be authenticated by the authorization server. Then, after obtaining a capability from the authorization server, the user presents the capability along with the request to the storage, allowing the storage to authenticate his access rights (specified in the $KeyData$).

Data privacy is needed whenever the data in transit or on storage must be protected from unauthorized disclosure. SNARE performs encryption and decryption at the client. Data are stored on storage in encrypted form, so it is protected from leaking by the storage (who does not know the

secret key since the *file-data key* is sealed in a *lockbox*) and there is no need to encrypt data again when it is sent over the network.

Data integrity is the means of ensuring that data modified on storage or in transit by a malicious intruder must be detected. SNARE ensures data and metadata integrity by verifying both block checksums and HMAC. Whenever an inode is read into memory, the HMAC will be verified to ensure the integrity of the inode information. Further verification of block checksums will ensure the integrity of data blocks, which is normally performed at the client during a read operation. Hence, any tampering on either the block checksum or the data block can be detected by users later. Note that a NAS in SNARE may attempt to mispresent data. For example, suppose a file has two data blocks b_1 and b_2. If a user requests b_1 and the NAS instead sends b_2 (together with b_2's checksum) back, then the user cannot notice it. This is because SNARE uses a single file-data key for all data blocks of the file. Alternatively, SNARE could use the same file-data key and include an initialization vector (e.g., block numbers) for each data block of the file to deal with such a malicious behavior. SNARE also prevents the *replay attack*.

Copy resistance is the means of guaranteeing that the data stored on a network-attached storage device could not be simply copied to another one with a different disk key. As described earlier, data block pointers are XOR-encrypted by a HMAC key, which is produced by hashing the *object ID* using the disk key K_d. So if an attacker can physically access a NAS disk [2] and copy the data from the NAS disk into another NAS with a different disk key, there is a difficulty to crack the data block pointer, thereby resulting in denial of data accesses.

3.3.3. File Sharing

The ability to share files among users is essential in a network file system. File sharing in SNARE is on per-file basis and depends on the concept of a lockbox. As discussed in Section 3.3.1, the *lockbox* in a file object's inode holds the file-data key and is read and written by a file-lockbox key. When an authorized user accesses a file on NAS (e.g., open a file for read), the *lockbox* is sent to the user. Since the user holds the corresponding file-lockbox key (which is sent by the authorization server along with the capability), he/she can read the file-data key, and then decrypts data blocks for subsequent reads or writes.

3.4. Basic Operation

In this section, we mainly discuss two basic operations provided by SNARE, namely data block write and data block read.

[2] We assume that its K_d is not disclosed.

3.4.1. Block Write

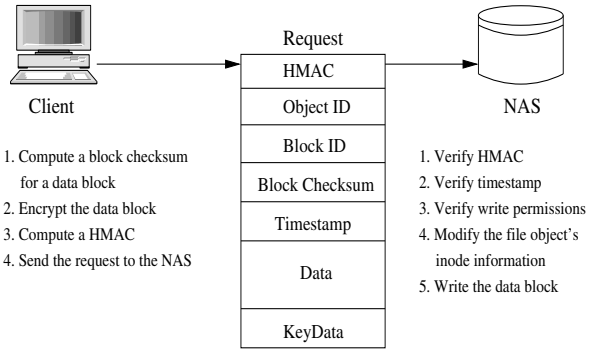

Figure 3. Writing a data block

As shown in Figure 3, to write a data block, the client calculates a block checksum over the data block, encrypts the data block, and computes a HMAC over $ObjectID$, $BlockID$, $BlockChecksum$, $Timestamp$, and $Data$ using a secret key hk_u. The client then sends the request to the storage.

Upon receiving the request, the storage either recalculates the hk_u using $KeyData$ and K_d or requests it from its cache, verifies the HMAC, checks the freshness of the request, and verifies the write permissions in $KeyData$. If everything succeeds, the storage stores the precomputed block checksum in the file object's inode, and writes the data block to the disk.

3.4.2. Block Read

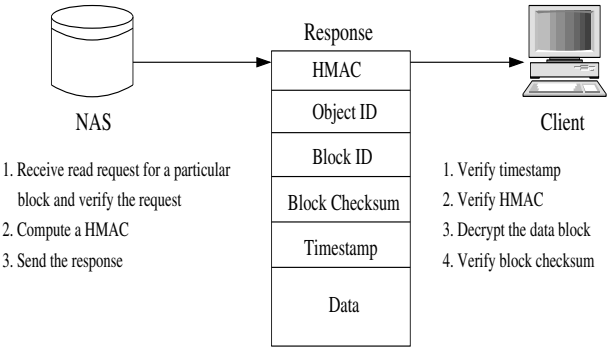

Figure 4. Reading a data block

For the read operation, SNARE takes advantage of the stored block checksum, which is precomputed at the client

during a previous write operation. As shown in Figure 4, the storage computes a HMAC over $ObjectID$, $BlockID$, $BlockChecksum$ and $Timestamp$ using the secrete key hk_u without including the data block. The amount of cryptographic work performed by the storage is therefore independent of the size of the data block transferred. Furthermore, the computation of the HMAC can be overlapped with the retrieval of the data block from the disk to minimize latency.

Upon receiving the response, the client verifies the response. The integrity of the data block is implicitly protected because of these two reasons: (1) the integrity of its block checksum can be verified by recalculating the HMAC, and (2) the client then recomputes a block checksum over the data block (which is decrypted by the *file-data key*) to finally ensure the integrity of the data block.

Compared to the storage, the client has to perform additional SHA-1 and decryption *over variable length data blocks*. The storage therefore can transmit secure data faster than the client is able to verify and decrypt the data, shifting the bottleneck from the storage to the receiving client. All the saved cryptographic work on the storage is achieved by the precomputed block checksum.

4. Cryptographic Protocols

In this section, we first present the protocols for user authentication. Then we describe a key distribution protocol by which users obtain their capabilities. Finally, we describe the request and response protocols by which the client communicates with the NAS. It is worth pointing out that all information in the protocols for both user authentication and key distribution are exchanged via a secure channel established between the client and the authorization server with session keys [17, 27].

Some notation is used in this section. Quoted values represent constants. K^{-1} represents the private key corresponding to the public key K. Subscript K represents a message encrypted with the key K, while subscript K^{-1} signifies a message signed by K^{-1}. $MAC_k(M)$ represents a HMAC by hashing message M using the secret key k.

4.1. User Authentication

When a user wishes to access the file system, he has to be authenticated by the authorization server first. In SNARE, the client daemon keeps a counter for each session and assigns a unique sequence number for each user authentication request.

In order to identify sessions uniquely, we define a $SessionID$ structure here:

$$SessionID = \text{SHA-1}(\text{“SessionInfo”}, K_{CS}, K_{SC})$$

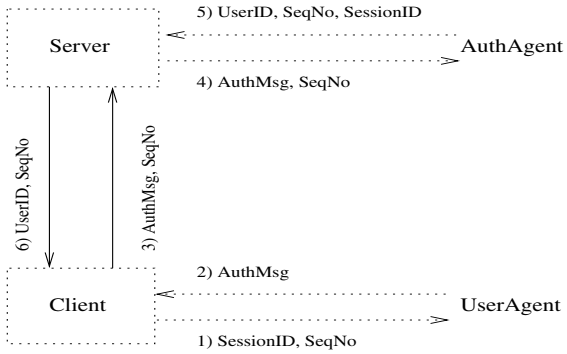

Figure 5. The user authentication protocol

Where K_{CS}, K_{SC} are session keys for the secure channel established between the client and the authorization server. They encrypt and guarantee the integrity of all communication in the session.

As shown in Figure 5, a user authentication request starts with passing a $SessionID$ structure and sequence number $SeqNo$ to the user agent by the client. The user agent returns an $AuthMsg$ by concatenating $SessionID$ with $SeqNO$, signing the result (using the user's private key ${K_U}^{-1}$), and appending the user's public key K_U:

$$\begin{aligned} SignedAuthReq = \quad & \{\text{“SignedAuthReq”}, \\ & SessionID, SeqNo\} \\ AuthMsg = \quad & K_U, \{SignedAuthReq\}_{{K_U}^{-1}} \end{aligned}$$

The client daemon appends another copy of the $SeqNo$ to the $AuthMsg$ and sends the request to the authorization server, which then forwards the request to its authagent. The authagent verifies the signature on the request using the user's public key K_U, and checks that the signed sequence number matches the one chosen by the client daemon. If the request is valid, the authagent maps the K_U to the $UserID$ by consulting one or more databases mapping public keys to user IDs, and returns the $UserID$ to the server together with the $SessionID$ and $SeqNo$. The server checks that the $SessionID$ matches the session and that the $SeqNo$ has not appeared before in the same session. If everything succeeds, the server returns the $UserID$ and $SeqNo$ to the client. After authentication, the client will tag all subsequent file system requests from the user to the authorization server with this $UserID$.

Since the entire user authentication protocol happens over a secure channel, it assures the authorization server that all authentication requests received must have been freshly generated by the client. The sequence number $SeqNo$ used here is not for security purpose. It allows the client to correlate the $UserID$ with the right user, because the client is

a multi-user machine.

4.2. Key Distribution

This key distribution protocol is used to keep the network-attached storage from having to perform key management, and allows a user to obtain a cryptographic capability from the authorization server when he accesses a file object for the first time. All subsequent accesses on this file object can reuse the capability unless it is invalidated.

In order to obtain a capability for a user, the client daemon constructs an authorization request $AuthReq$, which is sent to the authorization server:

$$AuthReq = \{\text{“UserAuthReq”}, UserID\ ObjectID, SessionID, SeqNo\}$$

Upon receiving the request, the authorization server first verifies the request. If the request is valid, the server generates a capability key data $KeyData$, which includes $UserID$, $ObjectID$, $Perms$ (access rights for read, write, etc.), $nounce$ (a number generated in a such way that the same number is not generated twice), $ExpTime$ (expiration time of the secret key based on the $KeyData$), and an access control version number AV. A user's secret key hk_u is generated by hashing $KeyData$ with a corresponding disk key K_d.

Finally, the authorization server sends the $AuthReply$ to the client:

$$\begin{aligned} KeyData &= \{\text{“KeyData”}, UserID, ObjectID, \\ &\quad Perms, nonce, ExpTime, AV\} \\ hk_u &= MAC_{K_d}(KeyData) \\ AuthReply &= \{\text{“UserAuthReply”}, KeyData, \\ &\quad hk_u, SeqNo, k_{lockbox}\} \end{aligned}$$

Note that the file-lockbox key $k_{lockbox}$ is included in the $AuthReply$. The sequence number $SeqNo$ used here is not for security purpose, and it allows the client to correlate the $AuthReply$ with the $AuthReq$.

4.3. Request

The client requests are of the form:

$$M = \{RequestArgs, RequestData, SeqNo, F_s\}, KeyData, MAC_{hk_u}(M)$$

The $RequestArgs$ field contains all the arguments being to specify a request (including operation type, etc.) except for data payload used for write operations (stored in $RequestData$). The $SeqNo$ denotes the sequence number, allowing the client to uniquely identify the request. The F_s denotes the NAS's timer, which is used to guarantee the freshness of the request. The $KeyData$ is associated with hk_u, and allows the NAS to regenerate the secret key hk_u using the disk key K_d. The integrity of $KeyData$ is implicitly protected because the modification of $KeyData$ will result in an incorrect HMAC being generated when the request is verified by the NAS. The $KeyData$ also allows the NAS to check the user's access rights on a file object by consulting the $Perms$. Moreover, the NAS can limits hk_u's lifetime by referring to the $ExpTime$ in the $KeyData$.

In particular, when the request is a write operation — writing a data block to the NAS, the client calculates a block checksum over the data block, encrypts the data block, and puts the checksum into $RequestArgs$. Then the client computes a HMAC using $MAC_{hk_u}(M)$ and sends the request to the NAS.

4.4. Response

The responses from the NAS to the client are of the form:

$$M = \{ResponseArgs, ResposeData, SeqNo, F_s\}, MAC_{hk_u}(M)$$

The $ResponseArgs$ field contains all the arguments being to specify a response (including the status of the corresponding operation, etc.) except for data payload used for read operations (stored in $ResponseData$). The presence of $SeqNo$ allows the client to properly correlate the response with its corresponding request. The F_s is included to compensate for clock drifts (as will discussed later). The $KeyData$ is not included in the response since the client already possesses the user's secret key hk_u.

In particular, when the response is for a read operation — retrieving a data block from the storage, it has the form:

$$M = \{ResposeArgs, SeqNo, F_s\}, ResponseData, MAC_{hk_u}(M)$$

Where, the data block is stored in the $ResponseData$, and the the data block checksum is contained in the $ResponseArgs$ field.

4.5. Freshness

Although the replay attack is not as powerful as the tampering attack, it can still cause serious damage to the integrity of a file system. Therefore, the storage device must be able to prevent the replay attack.

To ensure the freshness of messages, there are generally two alternatives to use: sequence numbers or timers. Sequence numbers require the NAS to maintain at least the

next sequence number and perhaps a list of other expected sequence numbers for each client. It will compete for memory with data caches and metadata caches. Since the NAS is relatively resource poor, it is not a good idea to use sequence numbers. Therefore, our system uses timers to let the NAS ensure the freshness of messages without keeping freshness information about all clients. As mentioned in section 4.3, storage devices require clients to include timer F_s in each request to ensure the freshness of the request.

The client synchronizes its timer with the NAS by keeping the difference between its timer and the NAS's timer. When the client sends a request to the NAS, it computes F_s using its timer and the difference, and then puts the F_s into the request. To compensate for clock drift, as mentioned in section 4.4, the NAS includes its current timer in all response, thus allowing the clients to resynchronize their timers each time the clients receive a response.

5. Performance Measurements

5.1. Experimental Testbed

At the time of this writing, we have built a prototype implementation of the cryptographic protocols. In addition, we constructed prototype network-attached storage devices and clients, and ran our experiments to see how much performance penalty the cryptographic overhead will impose on the storage. In this kind of experiments, the authorization server is currently not involved.

In our experiments, we used a DEC 333 MHZ Alpha with a Seagate Cheetah 10K RPM UltraSCSI disk drive as our network-attached storage device. The client ran on a DEC 500 MHZ Alpha. The storage and the client are connected to each other by 100 Mbits/s Ethernet using a switch. Our workload consists of reads and writes to logical blocks on the disk with random and sequential access patterns.

5.2. Cryptographic Overhead

Cryptography's computational requirements can have serious performance implications. In order to explore the overhead incurred by the cryptographic algorithms to the storage device used in our system, we tested the raw speed of SHA-1, RC5 encryption/decryption and RSA algorithm [22]. SHA-1 was tested for the cryptographic hash algorithm, RC5 was tested for the secret-key cryptographic algorithm, and RSA was tested for the public-key cryptographic algorithm.

As Figure 6 shows, the most expensive operation is RSA signature generation. Since we use a modulus of 512 bits with 32,767 as the public exponent in our RSA algorithm, this allows verification to be much faster than signature generation. SHA-1 is the fastest algorithm, while RC5 encryption takes less time than RC5 decryption. The amount of time required to compute a RSA signature justifies the avoidance of using public-key encryption on network-attached storage. Table 1 describes the cryptographic operations performed at the client and network-attached storage for each read and write request.

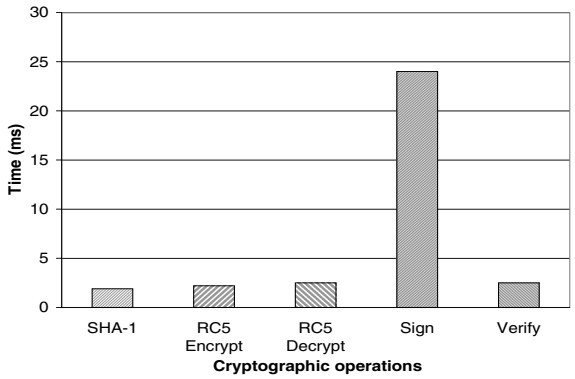

Figure 6. Performance of cryptographic algorithms on DEC 333 MHZ Alpha. Block size is 32 KB, while signature and verification are done on 128 bit inputs.

operations	Read		Write	
	Client	NAS	Client	NAS
Hash	√	√	√	√
En/decrypt	√		√	
Sign				
Verify				

Table 1. Cryptographic operations for each read and write request.

5.3. Prototype Performance

To illustrate the impact of security on performance, we first tested the system without any security, showing how fast the system could be with random and sequential access patterns. This is our baseline system. Then we tested the system with security enabled. Figure 7 shows the network-attached storage prototype's performance of reads and writes with/without security.

Figure 7 shows that, the performance of sequential accesses is much better than that of random accesses. In addition, with the block size varying from 1 KB to 32 KB, both random reads and random writes suffer little performance penalty for security. This suggests that random disk I/O op-

erations mean larger disk access time, therefore amortizing the performance penalty imposed by cryptography.

As mentioned in section 3.4.2, the storage exploits data block checksums to achieve good read performance. With precomputed checksums, the amount of cryptographic work performed on the storage is independent of the size of the data block transferred. Therefore, we have a fixed cryptographic overhead that is amortized over the size of the data block. As Figure 7 shows, with block sizes increasing up to 32 KB, the performance penalty of security decreases noticeably. For 2 KB sequential reads, the fixed penalty reduces performance by 32%, while for 32 KB sequential reads, the performance is reduced by only 9%. Large sequential reads ($\geq$ 4 KB) with security reduce performance by 9-25%.

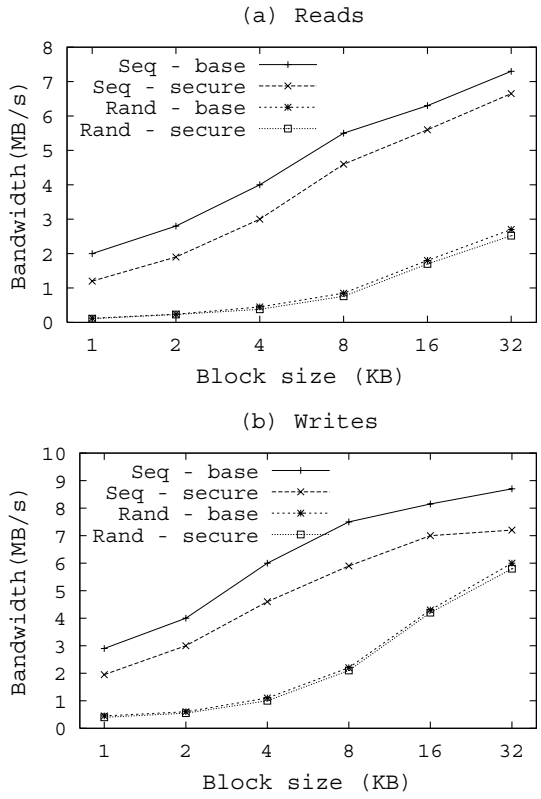

Figure 7. Performance of network-attached storage with/without security.

However, when the storage performs sequential writes, the amount of cryptographic work is dependent on the size of the data block transferred. Figure 7 shows that, large sequential writes ($\geq$ 4 KB) with security reduce performance by 14-23%.

6. Conclusions

Computer security is of growing importance in today's increasingly networked environment. In order to efficiently provide strong security on network-attached storage, we have to consider many aspects of the problem in order to decide how best to provide security while imposing small performance penalty.

This paper presents a strong security scheme for network-attached storage that is based on capability and uses a key distribution scheme to keep network-attached storage from performing key management. This system provides data privacy and integrity from the moment it leaves the client computer. In addition, it also guarantees that the data stored on the storage is copy-resistant.

Several methods are employed to boost the performance and scalability of network-attached storage devices. Using HMAC instead of more performance-intensive authentication methods such as public-key encryption and performing encryption/decryption at the client minimize the effort required by the network-attached storage device's CPU. Furthermore, with stored data block checksums (*which are precomputed at the client*), the storage can transmit secure data faster than the client is able to verify the data, thus shifting the bottleneck from the storage to the receiving client for good scalability.

In spite of this level of security, our experiments shows that, using a relatively inexpensive CPU in the storage device, there are little performance penalty for random disk accesses and about 9-25% performance degradation for large sequential disk accesses ($\geq$ 4 KB). Given the hostile environment in an untrusted, networked world, network-attached storage systems are able to provide strong security while preserving good performance.

7. Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments on drafts of this paper. We also thank Ning Shao for his help on our experimental setup.

References

[1] T. E. Anderson, M. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb. 1996.

[2] R. W. Baldwin and R. L. Rivest. The rc5, rc5-cbc, rc5-cbc-pad, and rc5-cts algorithms. Request for Comment (RFC) 2040, Oct. 1996.

[3] M. Blaze. A cryptographic file system for unix. In *Proceedings of the first ACM Conference on Computer and Communication Security*, pages 9–15, Fairfax, VA, Nov. 1993.

[4] M. Blaze. Key management in an encrypting file system. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 27–35, Boston, MA, June 1994.

[5] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the Freenix Track: 2001 USENIX Annual Technical Conference*, pages 199–212, Boston, MA, June 2001.

[6] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *CACM*, 9(3):143–155, Mar. 1966.

[7] W. E. Freeman and E. L. Miller. Design for a decentralized security system for network-attached storage. In *Proceedings of the 17th IEEE Symposium on Mass Storage Systems and Technologies*, pages 361–373, College Park, MD, Mar. 2000.

[8] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, June 1999.

[9] K. Fu, M. F. Kaashoek, and D. Mazires. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–196, San Diego, CA, Oct. 2000.

[10] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, Oct. 2000.

[11] G. A. Gibson, D. F. Nagle, J. B. K. Amiri, 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 Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[12] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, July 1999.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[14] H. Krawczyk, M. Bellare, and R. Canetti. Keyed-hashing for message authentication. Request for Comment (RFC) 2104, Internet Engineering Task Force (IETF), Feb. 1997.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, Cambridge, MA, Nov. 2000.

[16] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168–177, Cambridge, MA, 2000.

[17] D. Mazires, 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*, pages 124–139, Dec. 1999.

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

[19] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. strong secutity for network-attached storage. In *Proceedings of the 1st ACM Conference on File and Storage Technologies(FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

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

[21] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage sysytem security. In *Proceedings of the 1st ACM Conference on File and Storage Technologies(FAST)*, pages 15–30, Monterey, CA, Jan. 2002.

[22] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[23] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Nfs version 4 protocol. Request for Comment (RFC) 3010, Dec. 2001.

[24] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

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

[26] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, San Diego, CA, Oct. 2000.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[28] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, Aug. 2000.

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