Carnegie Mellon University Computer Science technical report CMU-CS-97-185, October 1997

Security for Network Attached Storage Devices

Howard Gobioff Garth Gibson Doug Tygar

October 23, 1997
CMU-CS-97-185

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents a novel cryptographic capability system addressing the security and performance needs of network attached storage systems in which file management functions occur at a different location than the file storage device. In our NASD system file managers issue capabilities to client machines, which can then directly access files stored on the network attached storage device without intervention by a file server. These capabilities may be reused by the client, so that interaction with the file manager is kept to a minimum. Our system emphasizes performance and scalability while separating the roles of decision maker (issuing capabilities) and verifier (validating a capability). We have demonstrated our system with adaptations of both the NFS and AFS distributed file systems using a prototype NASD implementation.

Sponsored by DARPA/ITO through ARPA Order D306, and issued by the Indian Head Division, NSWC under contract N00174-96-K-0002, Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, and DARPA under grant F19628-96-C-0061, the US Postal Service, and Toshiba Corporation. The project team is indebted to the generous contributions from the member companies of the Parallel Data Consortium. At the time of this writing, these companies include Hewlett-Packard Laboratories, Symbios Logic Inc., Data General, IBM Corporation, Seagate Technology, Storage Technology Corporation, and Digital Equipment Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the supporting organizations or the U.S government.

Keywords: capabilities, NASD, network attached storage, security

1 Motivation

1.1 A new direction for storage

The rapidly developing technology of peripheral storage devices has opened new opportunities for security technology [9, 19]. It is now possible to build a cost-effective disk drive that can directly communicate with the network without a server machine acting as an intermediary. These drives have minimal uncommitted computational power, so high-level file system functions (including access control) must still be handled by a separate server machine (which we call a *file manager*) that can understand local environment security needs [5, 10, 14, 11].

At Carnegie Mellon University, we are actively designing and building a system using disk drives directly connected to the network. The primary motivation for the NASD architecture is cost-effective, scalable performance. Directly attaching disks to the network can decrease the file server load [6] and increase the aggregate bandwidth available beyond what is available from a single storage server [7, 10].

File management is different in NASD systems than in classical distributed file systems such as Andrew File System [12], Sun's Network File System [23], or Novell Netware. In classical distributed file systems, a file server manages file storage on devices that are physically part of the file server machine (see Figure 1). In NASD, the disk drives are connected to the unsecure network, so the file management functionality must be partitioned between the drive and the file manager (see Figure 2) .

1.2 The Security Problem

But how can the file manager communicate access decisions to the drive in a scalable, secure, and efficient manner? Our paper addresses this question with a compact and efficient, distributed-capability mechanism. Briefly, a capability is a token which grants the bearer specified access rights. Unlike classical capability mechanisms [13, 20, 28] we cannot depend on all kernels being trusted to maintain system security in the face of a hostile network nor can we use a single machine with hardware support for capabilities. Our system will operate in a potentially hostile environment, with adversaries able to eavesdrop on network traffic and transmit arbitrary messages over the network. Even worse, we can not trust the client machine — since the user has physical control over the machine, he can insert arbitrary software, including kernel software, onto the machine. Thus, a client machine, under the control of a malicious user, may lie about its privileges.

We propose a novel representation for our capabilities; we use cryptographic capabilities issued by the file manager and checked by drives with minimal hardware support. The separation between issuing and verifying capabilities enables us to separate the file storage from file management — they may be done by machines separated by distance and with only indirect communication; access rights control is managed through the cryptographic information stored in the capabilities. These capabilities are presented by the clients when they attempt to access the disk drives.

This architecture introduces a number of exciting opportunities. Here are two advanced applications: permitting multiple file system to be built on top of the same drive interface; and storing drives in non-physically secure environments.

1.3 Advanced features: file system independent drive interface

Classical distributed file systems are implemented through one or more file servers which presented a particular distributed file system interface to users. NASD drives provide a fixed interface that is independent of the filesystem on which a file system is built. The independence from the file systems of the interface enables it to be used to build a stackable high-level file system which provides the interface and still use

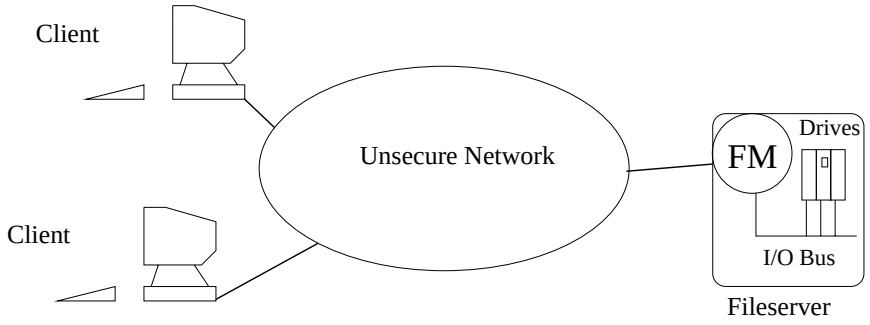

Figure 1: Classical distributed file system structure

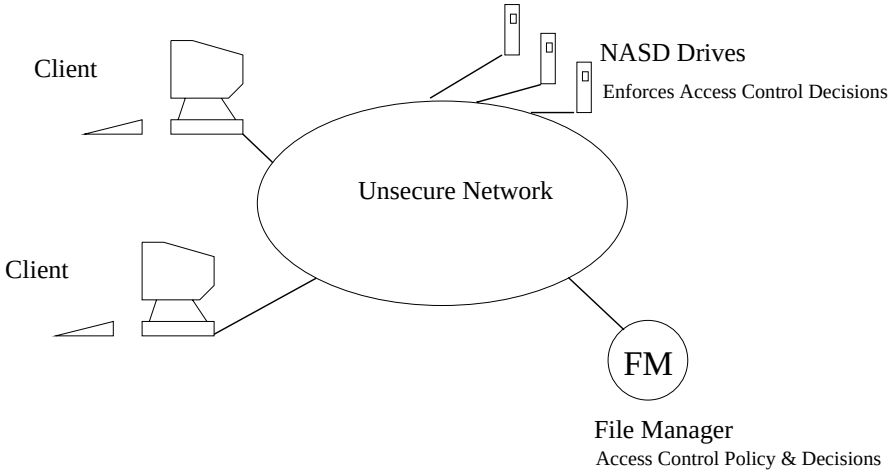

Figure 2: NASD distributed file system structure

a variety of file systems above the interface. For example, we can build a striping server that exports the NASD interface while striping data over a group of NASD drives. From the file system and user perspective, they are interacting with a single NASD drive while they are actually interacting with a striping server and a group of NASD drives.

Consider the options for an implementor of network attached storage devices. Existing storage and distributed file systems come in a variety of flavors. However, all file systems have both strengths and weaknesses that vary depending on the environment and the application. By enabling a variety of file systems to be built on NASD, we enable the selection of the best file system for the job and permit future development in the filesystem while using the same NASD drives.

1.4 Advanced features: disks without physical security

NASD presents us with an opportunity to use disk drives that can be stored in accessible or semi-accessible locations; we can relax the requirement that disks always be locked in physically secure locations. In classical distributed file systems, an adversary might attempt to steal information by first stealing the disk drive, and then reading it at his leisure. We normally prevent this type of attack by demanding that our distributed file systems be physically secure; with access only permitted to trusted individuals.

In modern working environments, for example small office or home environments; this requirement seems hard to achieve. With the spread of small WWW sites, we are increasingly seeing html, ftp, and file

servers being located in home offices or other unsecure environments where hostile physical access to the disk drives may be possible.

For drives that are physically secure, we can use the NASD structure to implement end-to-end encryption between the client and the drive. For the more challenging problem of drives that are not physically secure, we propose the use of a small amount of tamper-resistant hardware in NASD drives. This hardware can securely store keys used to encrypt files as they are stored and retrieved, without risk of exposure — even if physically attacked. All encryption, decryption, and re-encryption at the drive is executed under the shield of the tamper-resistant hardware. Moreover, because of our efficient capability mechanism, no file manager intervention is required for most file access. So our system can be secure, scalable, and highly distributed.

2 Overview

2.1 What is Network Attached Storage?

In NASD the file manager retains responsibility for administering the name space and access control policy of the file system but the file manager is bypassed in common case operations such as data transfer. The file manager makes the access control decisions according to the policies of the high level file system. When a common case operation, such as a data transfer, is requested by the client, the drive enforces the access control decision previously specified by the file manager. NASD uses a capability mechanism to communicate the file manager's access control decisions to the disk drive. What makes the NASD architecture significantly different from classical architectures is that policy decisions and enforcement are are separated by an unsecure network that introduces new security risks to data stored on drives.

Drives present the file manager and clients with a flat name space referring to variable sized objects in which to build the typically hierarchical file system users expect. Objects are the abstraction presented by the NASD disk drive; a file is an abstraction presented to the user by the file manager and client machine. At a minimum, a drive contains the components found in todays disks: a microprocessor, a network interface, buffer memory, and a storage device. Some drive implementations may include other hardware such as tamper-resistant hardware, cryptographic hardware support, or a (RAID) controller computer transparently coupling a group of simpler (SCSI) drives. High security drives may contain an additional secure coprocessor that is itself a general purpose processor and memory [29]. In some cases, drive designs will sacrifice some security by omitting tamper resistant hardware and cryptographic accelerators to reduce costs.

2.2 Options for security hardware

Different NASD drive configurations may provide users with different security guarantees. Here are some examples:

- A drive may provide authentication only on the control portion of message and not protect the data portion of requests. An adversary who is able to modify packets in transit could modify the data that is seen by the client or drive by acting as "a man in the middle".
- A drive may provide no privacy for user operations. Instead, the drive may only validate the integrity of operations. With this approach, an adversary could observe data transmitted over the network but would be unable to modify or forge requests. The distributed file system layered on top of NASD could provide privacy by storing encrypted data on NASD and performing its own key management to achieve sharing.

- Drives may encrypt data when it is sent over the network but store the data locally in unencrypted form. An adversary may be able to access data if he has direct access to the storage media or internal drive communication paths on the drive.

- By using a secure coprocessor, we can implement an encrypting filesystem and perform secure key management. With this approach, we can build systems where data is never available to an adversary as long as the coprocessor is uncompromised and the cryptographic systems are sound.

2.3 What does NASD require to be secure?

For a file system to run securely, the policy maker (file manager) and enforcer (drive) must exchange a minimum amount of information. When a file manager grants access to a user, the file manager must communicate the user's access rights to the drive. The drive must check that the information came from the file manager and is fresh. The goal of the NASD security system is to achieve this communication in a safe, flexible, and efficient way. One approach is for the file manager to inform the drive directly of a user's rights [14], which the drive will enforce when it later identifies the named user's requests. However, this approach requires that the drive be aware of the user's identity and implies a specific a user authentication mechanism to the drive. If the authentication mechanism is bound to the drive, an important degree of flexibility is removed from the design of the high level file system. Additionally, this requires the drive maintain information on a per user basis, which requires more state than a capability system.

An alternative, which we use in NASD, is to pass access rights to the drive through a cryptographic capability. The capability allows the authentication mechanism to be removed from NASD and a decision of the file system implementor rather than being dictated by NASD. Instead of being aware of the user's identity, the drive is only aware of a user's access rights. The capability arguments, which describe a user's access rights, and a capability key, which is used by the user to prove they are entitled to the rights they claim. They keys are derived from the capability arguments and a working key associated with the partition on which the object resides. Further details of the capability definition are given in Section 6.

How can the file manager efficiently and safely control access to the contents of a network attached drive? The file manager must be able to perform management functions on the drive and to control client access to storage. NASD's security infrastructure is designed to perform these functions while being resistant to attacks by an adversary with full knowledge of the NASD protocol.

Since out hypothetical adversary knows our protocol, the ultimate security of the system rests on keeping the keys private (ie. key management). The NASD key management hierarchy balances flexibility of use and security. We support key management that can take advantage of tamper resistant hardware when it exists, and in other cases effects key management through software control.

2.4 Brief description of the Key Hierarchy

NASD uses a hierarchy of four kinds of keys: a master key (*MasterKey*), a drive key (*DriveKey*), partition keys ($PartionKeyN$ where N denotes the partition), and working keys (black Kb_N and gold Kg_N). The key management issues and key hierarchy is discussed in Section 7.

- *MasterKey* : An administrator uses the master key to control the key hierarchy and for other administrative tasks. The *MasterKey* is an immutable key so should be used infrequently and be protected. Using the *MasterKey*, an administrator can set the *DriveKey*.

- *DriveKey*: The *DriveKey* manages the partioning of drive capacity and delegates authority for entire partitions to a file manager by setting up partitions and *PartitionKey*s.

- *PartitionKeyN*: File managers use the *PartitionKey*s to manage a partition and to set the *WorkingKey*s that are used for the most frequent operations between the file manager and drive.

- *WorkingKey*: The file manager uses *WorkingKey*s to generate capabilities. Each drive maintains two working keys per partition. Section 7.4 explains why two working keys are necessary.

File managers enable client access to storage by providing clients with a capability to access a stored object. By presenting the capability to the drive and generating a keyed message digest of the request, the client can convince the drive that the operation is authorized (Section 6 discusses capabilities) because of special properties of the capability key.

To check the freshness of messages and impose a finite lifetime on capabilities, NASD drives have secure clocks which generate monotonically increasing timestamps at a constant rate. A drive's secure clock is protected by the physical security guarantees of the drive. The timestamps enable a drive to recognize a message as either too old or dated in the future (Section 9 discusses clocks). A filesystem's timekeeping needs may or may not be also met using the secure clock. Our design allows for a real-time clock to be maintained separately.

NASD's keys and capability mechanism subsume file-drive authentication and client-drive authentication. Authentication is still necessary, but the exact method of doing it is left to file system designers. The NASD design is not locked to a single authentication method.

This paper presents a design of a security system for use in network attached storage devices. We focus on the portions of the system fixed by the design of the storage devices and leave higher level issues to file system implementors.

3 Design Goals

We had the following goals when designing the security for NASD:

- The NASD security system should explicitly separate the policy enforcement mechanism from the policy decision process (a similar goal was stated for Hydra [28]) so we the file manager must be able to communicate policy decisions to the drive.

- The protocol should prevent unauthorized modification of client requests and capabilities along with protecting privacy of requests if dictated by the policies of clients or file managers.

- To minimize interaction with the file manager, the drive should be able to validate client operations without direct communication with the file manager.

- To allow for low memory drive implementations, there should be no long term state shared between drive and client. Overall state requirements of the drive should be kept at a minimum, but additional memory should enhance performance.

- The security protocol should add as little overhead as possible in terms of computation and the number and size of messages.

4 Notation

- The user (*U) may be a software agent, a person, a smart token, or anything else that can convince a filemanager to provide access rights.*
- *The file manager (*FM) is responsible for making policy decisions regulating access to objects stored on NASD drives.
- $E_K(Msg)$ is message encrypted under key *K* using symmetric cryptography.
- $MD(Msg)$ is a message digest (cryptographically strong hash) of *Msg*. Various secure hash algorithms such as MD5 [22], SHA [18], or Tiger [1] may be viable options for NASD.
- $MD_K(Msg)$ is a keyed version of a message digest (also called a MAC or message authentication code). Constructions such as HMAC [2] may be good candidates to construct a keyed digest based on an unkeyed version. Karp-Rabin [21] may also be viable since it is inherently keyed.
- $K_{(A,B)}$ is a temporary (session) key shared between entities A and B.
- AV is a version identifier on the per-object access control information that the file manager controls. The *AV* is the only security related information that a disk must maintain on a per object basis. Section 8 discusses the *AV*.

5 NASD Requests

All common operations on a NASD drive, regardless of origin, have the same basic format. Requests are of the form:

$$KeyType, ActualProtection, CapabilityArgs, RequestArgs, RequestData, \textit{Timestamp-Nonce}, Digest$$

The *KeyType* field specifies what key the request is being authorized under. This could be the master key, the drive key, a partition key, a working key, or a capability. When a capability is being used, the *KeyType* will indicate which working key was used to generate the capability; we call this the basis key. Section 7 discusses the key hierarchy and Section 6 discusses capabilities.

5.1 Protection options

The *ActualProtection* field describes the protection options used for this request. The protection options for a request are a combination of flags indicating:

1. **IntegrityArgs**: The integrity of *RequestArgs* and *Timestamp-Nonce* are protected by *Digest*.
2. **IntegrityData**: The integrity of *RequestData* is protected by *Digest*.
3. **PrivacyArgs**: The privacy of the *RequestArgs* is protected through encryption.
4. **PrivacyData**: The privacy of the *RequestData* is protected through encryption.
5. **PrivacyCapability**: The privacy of the *CapabilityArgs* is protected through encryption.

Some combinations, such as **IntegrityData** without **IntegrityArgs**, do not make sense. If no options are enabled, the drive will check that the capability arguments are valid for the request but will be unable to detect modification of the request or a forged request (this is similar to NFS). With **IntegrityArgs** and **IntegrityData** enabled, the drive protects the request from any modification. Manipulation of drive configuration information must be protected with at least **IntegrityArgs** and **IntegrityData** to maintain the integrity of the system. Key management operations require **PrivacyData** as well.

Each drive partition specifies the minimum acceptable security options for its requests. If the minimum acceptable security options do not include **IntegrityArgs**, then anyone can generate a request by setting *ActualProtection* to no options; therefore **IntegrityArgs** must be used if the file manager wants to implement any security policy stronger than no security.

Section 6 discusses *CapabilityArgs* and how they are used.

5.2 Remaining fields in the request

The *RequestArgs* field contains all the arguments being to specify a request (including operation type) except for the variable length data payload used for read or write operations (stored in *RequestData* or the equivalent in the reply). The *Timestamp-Nonce* is used to protect against replay or delay attacks.

A reply is similar to a request but omits the *KeyType*, *ActualProtection*, and *CapabilityArgs* because they are assumed to be the same as in the request. The reply's *Timestamp-Nonce* must match the request's to allow the receiver to properly correlate requests and replies (while the underlying RPC or messaging system may also do this, lower level support is not protected by the security of NASD.)

The *KeyType* and *Actual Protection* are not explicitly protected by any of the protection options. These fields are used to select what key to use in processing the request and how to check the security of the request. The integrity of *KeyType* and *ActualProtection* is implicitly protected if either **IntegrityArgs** or **IntegrityData** is used because the modification of either of the fields will result in an incorrect digest being generated when the request is verified.

6 Capabilities

6.1 Construction

The heart of the NASD security system is the capability architecture governing access to NASD objects. A NASD capability is the capability arguments and a capability key. The capability arguments are a tuple containing:

- Drive identifier
- Partition
- Object identifier
- Region (byte range)
- Access rights
- Expiration time
- Minimum protection requirements
- Basis Key (black or gold)

- Audit identifier

The capability key is a cryptographic key used by a client to bind a specific NASD request to a capability through used of a MAC. The MAC creates a relationship between the fields of the capability, the request, and the digest. If the drive generates the same MAC that was included in the request, the drive can be confident that the MAC, the capability arguments, and the request were all received without tampering.

The capability key is $MD_{Kx}(\textit{CapabilityArguments}, AV)$ where Kx is either the black or gold key for the partition. The dependence on the *AV* enables revocation and the the dependence on Kx means only the file manager or drive could generate the capability key.

The first four fields identify the set of bytes for which the capability is valid. The *access rights* and *expiration time* limit the operations and time range for which the capability is valid. The lifetime of a capability can not exceed the lifetime of the working key under which it was issued, otherwise the validity of the capability can not be verified. The *basis key* indicates if this capability was issued with the black working key or the gold working key which is used to generate the capability key. The *minimum protection* indicates a floor set of security options that the file manager requires of the client for all operations using the capability. This permits the file manager to require a higher standard of security than is required by the partition.

Finally, the *audit identifier* is a field where the file manager can associate arbitrary values with the capability. In auditing implementations of NASD, the drive will record the *audit identifier* in a log record for an operation performed with the capability. This allows a log-analyzer to later extract this information from the logs on the drive. With the *audit identifier*, we can easily integrate drive security with other applications such as transaction management (by recording a transaction ID in the *audit identifier*) or audit trails (by recording the users identity in the *audit identifier*).

NASD requires that the file system specific client-file manager protocol provide private transmission of capability keys to the clients along with transmission of the capability arguments. We expect this will be performed as part of a directory lookup process, but that is not a requirement for NASD. The capability arguments are necessarily kept private. If an adversary can read the capability arguments, he still can not access the object without a working key or the capability key. If an adversary modifies either the request or the response, he will at best execute a denial of service attack. However, if the adversary can perform such an attack, he can probably interfere with any network traffic — a more general denial of service attack.

A client can prove it has the right to access an object by using a capability key to encrypt and/or digest a request. The capability keys are derived from the capability arguments and a working key for the partition where the object resides. When the capability arguments are given to the drive, the drive can independently compute the capability key and verify the critical parameters for an access check.

If the capability arguments and capability key don't match, then the drive will be unable validate the request. Since clients know neither of the working keys, clients are unable to generate a forged key. If the *AV* or working key changes then the drive will also be unable to generate the expected capability key. However, in this case, the changes correspond to expected invalidations of access rights.

6.2 Use

NASD drives may accept encrypted messages, messages with a MAC, or both. Encryption performance can be drastically reduced by using hardware support [4] and the MAC can also be implemented in hardware.

The MACing or encryption of data is the basis for a generic signature function $S_k(\textit{Msg})$ which is used extensively in the NASD security system. In the case of clear-signing,

$$S_k(\textit{Msg}) = (\textit{Msg}, MD_k(\textit{Msg}))$$

while when using encryption and MAC:

$$S_k(\textit{Msg}) = E_k(\textit{Msg}, MD_k(\textit{Msg}))$$

Performing the message digest before encryption is important because this prevents an adversary from modifying *ActualProtection* to corrupt data by removing the privacy option when both privacy and integrity are being used. If the message digest were performed second, an adversary could remove the privacy option and cause the request to be accepted but have the encrypted data written out – effectively trashing stored data.

A client's request to a drive for **Operation** looks like the following (the format discussed in Section 5):

$$KeyType, ActualProtection, CapabilityArgs, S_{\textit{CapabilityKey}}(\textbf{Operation}, \textit{Timestamp-Nonce}), UnprotectedOperation$$

Note that part of the operation is unprotected by the generic signature. Some of the data may be sent without a digest and unencrypted for performance reasons although this reduces the security of the interaction. The NASD protocols support a variety of combinations.

6.3 Request Validation

To validate a request, the drive must obtain the capability key that generated the signature. The drive can either recalculate the capability key or request it from an on drive cache of recently seen capabilities. To maintain correctness, the cache should not return capabilities that are invalid such as ones dependent on out-of-date data that has been updated by the file manager (e.g. *AV* or working keys). To recalculate the capability key for a request, the drive needs all the fields used in the initial key construction. The drive has the working keys associated with a partition and the *AV* associated with the object but the client must provide the capability arguments. The keys are maintained in key storage on the drive and the *AV* is part of the object metadata information. Because of the dependency on the working keys and the capability arguments as discussed in Section 6.1, any modification of the capability arguments will be detected because the wrong capability key will be generated.

After the drive checks that the generic signature matches the request, the freshness and uniqueness of the request must be checked. The drive should reject messages that have been delayed (or replayed) beyond a certain timeout threshold. We address the replay problem by adding tags to message that are valid exactly once. To reduce storage and messaging requirements, we use unique timestamps as nonces.

We require that the *Timestamp-Nonce* be approximately accurate (enough so that it will be within a fixed tolerance ϵ of the current time when it arrives at the drive) and that the *Timestamp-Nonce* be unique. We maintain a counter of NASD messages sent within the current clock quantum and use the counter to set the low bits of the *Timestamp-Nonce*. A *Timestamp-Nonce* is generated fresh for each request to guarantee the uniqueness for the previously mentioned security check.

At this point, the drive has verified that the request is bound to the capability, the capability is valid, and that the request is both fresh and unique. The final step is for the drive to verify that the operation is permitted by the access rights described in the capability. The set of primitive operations and the representation of access rights are not explicitly defined in the security protocol. At first approximation, the access rights could be represented as a bit vector where each bit indicates a primitive NASD access mode (such as read and write) is permitted. However, a more complex representation with more information is also possible.

6.4 Replys

The reply from the drive to the client will be of the form:

$$S_{\mathit{CapabilityKey}}(\mathit{ReplyBody}, \mathit{Timestamp\text{-}Nonce}_{\mathit{Request}})$$

where *ReplyBody* has the status and results of the operation. The underlying RPC layer will relate the reply back to the original request so that the requestor knows what key to use to verify the request.

Since the reply is signed with the capability key the client can verify that the message came from the drive (assuming the file manager is behaving properly and there have been no security compromises). The inclusion of the $\mathit{Timestamp\text{-}Nonce}_{\mathit{Request}}$ enables the client to correlate this response with correct request and detect replay attacks. While the layers on which the requests and response are built, such as an RPC mechanism, may provide the same functionality, the values will not have integrity guaranteed by $S_{\mathit{CapabilityKey}}$ and are therefore subject to spoofing. It is important to note that the *ReplyBody* and *Operation* must be different to prevent an adversary from sending the signed *Operation, Timestamp-Nonce* tuple as the reply.

7 Key Management

7.1 Initialization

When a drive is shipped from the factory, the owner needs some way of initializing it with key information, so that only authorized users can subsequently exercise control over the drive. Initially, master and drive keys are set to the serial number of the drive, no partitions exist, and the drive is in an *uninitialized* state. Immediately on receipt of a drive the owner should move the drive into an *initialized* state by sending it an **Initialize** message. An **Initialize** message will set the maser and drive keys and move the drive from an *uninitialized* state to an *initialized* state. Once a drive has been initialized, *the master key can not be changed*.

We perform the initialization by sending the drive the following message:

$$\textbf{Initialize}, \mathit{NewMasterKey}, \mathit{NewDriveKey}, CurrentTime$$

The **Initialize** message must be sent over a trusted channel, such as an isolated administrator's network, to avoid eavesdropping. While this could be cumbersome to administrators, the process could be automated except for the physical connection/disconnection of the drive. The other keys are not set by the **Initialize** message because they are associated with specific partitions and no partitions exist when the drive is initialized.

The trusted channel could be as simple as a single network link from an administrator's machine to the drive. If an adversary can eavesdrop on the **Initialize** message, they would have the master key and unrestricted access to the drive so we recommend that the drive be initialized while it is under the physical control of the administrator — for example, while the administrator can physically observe the integrity of the cable going from the drive to the administrator's notebook computer. No encryption or message digests are used; access to the drive in the *uninitialized* state is proof we are permitted to perform the initialization procedure. When the drive is in an *uninitialized* state, the drive assumes that **Initialize** messages are valid. Once the drive receives an **Initialize** message, it rejects all subsequent **Initialize** requests. (The **Initialize** message does not follow the standard convention for NASD messages discussed in Section 5).

7.2 Master Key

Since the master key is, by assumption, immutable, we must be very careful to limit the exposure of the key. The master key should not be kept online and it should never be exposed to the file manager. We must also limit the amount of text encrypted or digested under the master key; the master key is used to change the (mutable) drive key.

The master key is used to set the drive key when:

- The drive key is known or suspected to be compromised.
- Failures cause the file manager and drive to have different drive key values.
- A regular (infrequent) key change is scheduled to maintain security.

To change the drive key the following **SetDriveKey** message is sent:

$$UseMasterKey, ProtectionOptions, \{\}, SetDriveKey, NewKey, RequestNonce, RequestDigest$$

where $ProtectionOptions \supseteq \{\mathbf{IntegrityArgs}, \mathbf{IntegrityData}, \mathbf{PrivacyData}\}$.

This message enables someone holding the master key to change the drive key. The message will only be accepted if at least the protection options of $\{\mathbf{IntegrityArgs}, \mathbf{IntegrityData}, \mathbf{PrivacyData}\}$ are used in the request and the key type is *UseMasterKey*. Similar operations exist for other keys in the hierarchy except the *MasterKey*. A key change request must be authorized by the key immediately above the key being changed (see Figure 3).

7.3 Partition Keys

The partition keys are regularly used to change the working keys and will likely be kept on the file manager so they are exposed to potential compromise. A compromise of the file manager would expose resources on the drive protected by the partition key. However, depending on the implementation hardware, we can minimize the risk to the keys by utilizing a secure coprocessor on the file manager to protect the partition keys.

7.4 Working Keys

The drive key is usually not on a file manager to help isolate the security concerns of different file managers managing different partitions on a drive. The inability to access the drive key implies that that file manager can not change the partition key for regular updates.

However, NASD capability system requires a key to generate capabilities. This key must be regularly changed to limit exposure of the key. The need to regularly change the key motivates the addition of the working keys to the key hierarchy. The working keys for partition *N* are the black key (Kbl_N) and the gold key (Kg_N). The black and gold keys are changed by the **SetBlackWorkingKeyN** and **SetGoldWorkingKeyN** message (similar to the **SetDriveKey** message) at whatever interval the file manager requires (we suggest at least a daily update).

If we maintained only a single working key, the update of the key would automatically invalidate all outstanding capabilities. Two working keys avoids this bulk invalidation (and prevents the subsequent storm of requests for new capabilities) because a file manager can phase out an old *WorkingKey* by issuing capabilities with the new working key while capabilities in the old working key gracefully expire.

The construction of a capability is directly dependent on the value of either the black or gold key (see Section 6). If a file manager wishes to invalidate all issued capabilities, the file manager needs to change both the black and gold key.

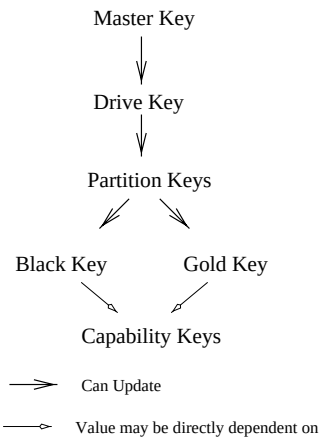

Figure 3: Key Hierarchy Relationships

7.5 Key Protection

Clearly, protecting keys is central to maintaining security of our system. A limited level of protection can be achieved by locating the drives and the file managers in physically secure facilities with controlled access. Security can be further enhanced by using secure (tamper resistant) coprocessors on the file manager and drive to perform key management. Encapsulating the key management in tamper resistant hardware complicates the task of an adversary attempting to search the internals of the drive for secret keys contained. If operations involving keys are performed underneath the tamper resistant hardware shield, keys need never be exposed.

7.6 Drive Management

When a drive is transferred to a new owner, it must be reinitialized. If a user sends a **Reset** message to the drive authorized by the *MasterKey*. The **Reset** message causes the drive to return all keys to the factory setting and destroy all data stored on the drive. We note that some critics claim that because of magnetic memory effects, magnetic media can be subject to data recovery by skilled adversaries even when the data is erased and written over. If the data is stored in unencrypted form, the data may always be recoverable until the media is physically destroyed.

In the event that the *MasterKey* is compromised, we treat this as a disaster scenario and consider the contents of a drive as lost. The drive must be **Reset** and data recovered via RAID or a backup mechanism.

8 Access Control Version Numbers

The access control version number (*AV*) is maintained both at the file manager and the drive. The *AV* enables the file manager to invalidate outstanding capabilities when access control information for an object changes. Of course changing the *AV* invalidates all outstanding capabilities for an object, so it is a coarse tool. We expect capabilities will be invalidated primarily when the access control information changes to become more restrictive.

The drive must differentiate between valid and invalid capabilities but the drive lacks sufficient resources to maintain a record of currently valid and invalid capabilities. Another option is to maintain a blacklist of

invalidated capabilities, but this list could be large if many capabilities are revoked. Therefore we use an AV to allow revocation of capabilities by notifying the drive that access control information has changed.

9 Time

In order to detect replays as described earlier, the client and drive must share a value for the current time. Since timestamps must be unique, we require that this time always be monotonically increasing. In a real world environment, the NASD object system will also need a real-time clock that can be adjusted for skew, system failures, daylight savings time, or time zone changes. We recommend that a drive should have a physically secure clock that is monotonically increasing and separate from a more flexible real-time clock.

The real-time clock is used for file system semantics such as file modification times. The real-time clock can be maintained as an offset from the secure clock or may be maintained differently depending on the implementation.

When a client communicates with a drive or file manager, the client must know the proper (secure) time value for generating timestamps. In the case of client-drive communication, the client does not share any long term shared state that can be used to authenticate time requests. NASD requires that the file manager know the current value of the drives clock. This could be achieved by an authenticated clock synchronization protocol (perhaps similar to SNTP [16] or NTP [15]). Similarly, the client should synchronize with the file manager on the secure clock value of the drive. The file manager-drive time synchronization can use the daily key; the client- file manager can use the session key.

10 Prototype

10.1 Experimental Testbed

We have built a NASD prototype implementation of a drive and adapted NFS and AFS to run with NASD drives. Our prototypes run on DEC Alphas running Digital Unix 3.2. The file system clients and the drive prototype all run inside the kernel while the file managers run at user level. All communication with the drive uses DEC's DCE RPC package. In the NFS prototype, the client-file manager communication also uses DCE because NFS's simple file manager was written from scratch. In the AFS prototype, the client-file manager communication uses RX, which is the normal AFS protocol, because the more complex file manager needs to communicate with other unmodified AFS servers[1]. Our experimental security results are for the NFS prototype, although the drive prototype is identical for both the AFS and NFS prototypes.

For our experiments, we used a DEC Alpha 3000/400 with a HP 2247 drive as our NASD drive. The file manager ran on a DEC Alpha 3000/400 and the client was a DEC Alpha 255. All the machines were connected by ATM through a DEC 155 Mbps (OC3) ATM Gigaswitch.

The prototype supports the **IntegrityArgs** and **IntegrityData** protection options. We implemented the security as a wrappers around the NFS client RPC stubs and drive functions. Our use of keyed message digests for key generation and authorization requires that the security layer of the system be able to access RPC arguments and data in unmarshalled form, so the fields may be constructed and processed, and marshalled form, so the data is in a canonical form and will generate the same digest at either end. This requirement complicated our use of DCE and forced us to write our own marshaling code instead of utilizing the marshaling available through DCE.

[1] An AFS cell is actually several servers working together to manage protection, volumes, and other functions. All these servers communicate using RX.

We use a software implementation of HMAC-MD5 as our keyed message digest. Our MD5 implementation is derived from the cryptolib implementation from AT&T Labs. The MD5 code was modified to use many of the optimizations described in [26] to avoid unnecessary byte swapping or copy operations. The HMAC layer above MD5 is the reference implementation of HMAC.

10.2 Description and result of experiment

Our basic experiments helped us under the performance cost of using the security in NASD. Our experiment was to read 67MB of data in 8192 byte blocks off the drive to measure sustained bandwidth. All caches were flushed before the experiments began. This experiments provides an indicator of the cost of security when using NFS (since NFS is restricted to 8192 byte requests). All numbers reported are the average of 5 runs.

Initially, we performed the experiment without any of the security options enabled and achieved an average bandwidth of 27.5 Mbits/second. When **IntegrityArgs** was turned on, our bandwidth reduced by 18 percent to 22.5 Mbits/sec. By also turning on **IntegrityData**, the bandwidth dropped more dramatically by another 19 percent to 17.4 Mbits/sec. The expense of even doing a HMAC-MD5 on part of the data had a noticeable impact on performance which motivates our interest in hardware support for security. These numbers were with a capability cache enabled so the capability key was only computed on the first request from the client in each experiment. Subsequent requests were able to use the cached capability key.

11 Related Work

Historically, capability systems have used hardware support to prevent client modification of capabilities [3, 27] or depended on trusted operating system kernels [24, 28]. NASD capabilities differ because we use a fully distributed system where we do not trust the holders of the capabilities. While the basic idea of a capability in these examples and NASD is essentially the same, our inability to trust the holder of the capability and the distributed nature of NASD differentiates our problem from past systems. Instead NASD generates and verifies capabilities using a known protocol and secret keys shared between the issuer and validator of capabilities.

Both ICAP [8] and Amoeba [25] utilize one way functions in capability construction. However, neither ICAP nor Amoeba explicitly separate the issuer of a capability from the validator of a capability.

The proxy model [17] is a more powerful abstraction than capabilities. The emphasis in the proxy model work has been the power of the abstraction and integration with Kerberos rather than an explicit emphasis on performance. However, Neuman advocates that the mechanism be built on top of the authentication mechanism. Since NASD file systems may use a variety of authentication systems all using a single drive protocol, NASD requires that the authentication mechanism be separate from the drive authorization to enable different file system implementations.

The NetStation project [14] is another network attached storage system. NetStation's storage model is a block device attached to the network. Since the storage only provides a block model, the meta-data to provide a file abstraction must be shipped to the drive for each file that is opened. The file manager sends a message to the drive that explicitly "grants" a user access to an object (see Figure 4) and provides the drive the the file metadata. A pre-access permission request from the client to the file manager causes the file manager to engage in a "grant" message exchange with the drive. Unfortunately, this requires the drive to maintain more state in terms of outstanding grants, adds to the drive-manager messaging, and increases the total messaging done by the NASD system.

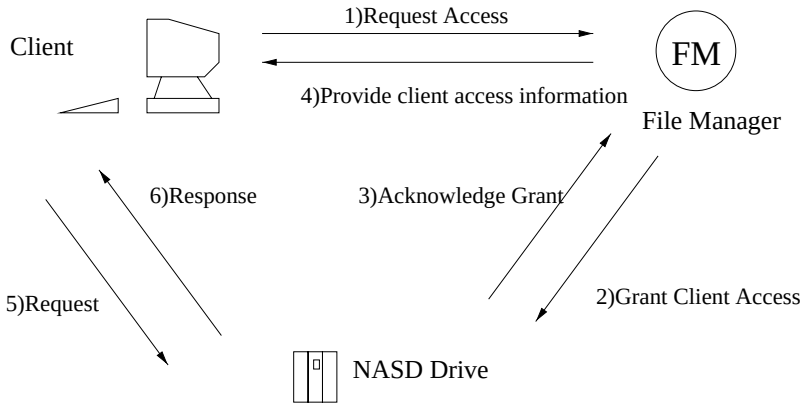

Figure 4: Message flow using grant-style approach

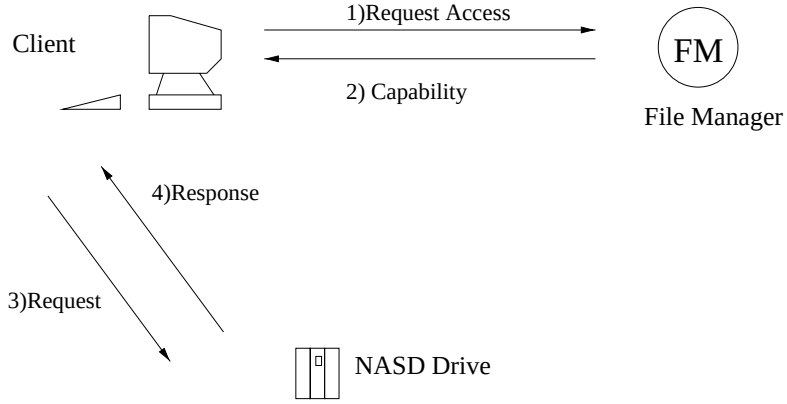

Figure 5: Message flow without grants

The NASD infrastructure discussed in this document eliminates the "grant" message exchange (see Figure 5) by using the capability system described in this paper. By including all the authorization information in each request, we eliminate the need for the file manager to explicitly inform the drive of privileges granted to a client. In both the NetStation and NASD the, the same information is being communicated to the drive but the path of the information differs. Where the grant model (NetStation) has the file manager informing the drive of user access rights, the capability model (NASD) passes this information back through the client. By removing the grant message and reply, we reduce the message count on an initial access by two messages resulting in lower latency to the client.

Both NetStation and NASD require the file manager to interact with the drive when access control information is changed. With the grant model (NetStation), the file manager must explicitly revoke previously granted access rights to meet the new access control restrictions. In the capability model (NASD), the file manager informs the drive that an object's access control information, *AV*, has changed. This effectively revokes all outstanding capabilities. So, NetStation permits a finer degree of revocation than is available in NASD. Therefore, the grant/revoke model may be more efficient when frequent and fine grained revocation is performed.

12 Conclusion

The NASD architecture is an innovative approach to the problems of high performance and cost effective I/O based on network attached storage systems. By providing security to network attached storage, we enable clients to utilize the potential performance and scalability benefits inherent in network attached storage without compromising their data security.

The essence of our capability scheme is the encapsulation of the bearer's access rights on a particular version of a storage object using a secret key shared between capability issuer (file manager) and capability enforcer (NASD drive). In fact, the secret key is one permanent and 4 periodically changing keys forming a key hierarchy. While changing keys can be used for capability revocation, a finer grain of selective revocation is possible through changing the access control version number on a NASD object. Because the encapsulated rights in a capability are also disclosed to clients, they can be delivered to the NASD drive with each request allowing capability verification without per client state in each drive. For higher performance, a drive can cache computed capabilities.

On top of the protocol given in this paper, an arbitrary distributed file system access control model can be imposed. With the file manager making policy decisions, the fact that the file manager and disk communicate via capabilities does not prevent the file manager from supporting a UFS user-group-world model, a AFS ACL model, a NFS UID/GID model, or another model.

In order to permit the implementation of arbitrary distributed file systems, NASD system separates the policy decision role, the file manager (which embodies the distributed file system personality), from the policy enforcement, the NASD drive. To support this separation of roles, the NASD capability system similarly separates the issuing of a capability from validating a capabilty. Using the capabilities in NASD, a file manager can securely communicate authorization decisions to a NASD drive.

12.1 Future Work

Our immediate future work is focused on increasing the performance of NASD while reducing file manager load. Caching previously computed message digests to speed up later reads and asynchronous security error notification for writes are promising directions.

Additional work is being done in redefining what a capability refers to. Currently, a capability refers to a single object but we expect to get performance benefits out of a more powerful naming mechanism that allows NASD capabilities to refer to sets of objects.

We are also investigating the use of different types hardware support for security to improve both functionality and performance. In high-security, high-performance environments, hardware assistance for the cryptographic functions will prove invaluable. However, drives are cost sensitive and the cost of such security will not be appropriate for all environments.

Looking further out, we believe that our architecture will find applications beyond file storage. For example, it is interesting to consider extending the NASD capability system to other network appliances, such as cameras or remote displays, in a distributed system. The capabilities could also be used to protect other network resources such as databases or video servers. This has applications in mobile code applications — for example, an instructor may want to gain temporary access to a large classroom display during her class. We believe that it is possible to adapt NASD protocols to realize such an application.

Another long-term extension of the NASD security system is to consider the use of smart tokens with tamper resistance (such as smart cards) that implement the the NASD protocols. Such a system would allow a user to easily allow a fixed client machine (such as workstation in a hotel room or a temporary office) to gain access to the user's files; later, when the smart token is removed, the capabilities and rights are also removed. While a client machine with Trojan horse software would be able to potentially keep a record

of data sent to that machine, the machine would never have any long term security information that would permit continued access. These smart cards would be most useful if they could access multiple file systems and multiple client operating systems. Designing a generic interface that works securely and efficiently with all combinations of client OS and distributed file system structure is a significant challenge.

References

[1] Ross Anderson and Eli Biham. Tiger: A Fast New Hash Function. In *Proceedings of the Third Workshop on Fast Software Encryption*, number 1039 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In *Advances in Cryptology: Crypto '96 Proceedings*. Springer-Verlag, 1996.

[3] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogramed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[4] Hans Eberle. A High-speed DES Implementation for Network Applications. Technical Report 90, DEC System Research Center, September 1992.

[5] Garth Gibson. Secure distributed and parallel file systems based on network-attached autonomous disk drives - White Paper, September 1995.

[6] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File Server Scaling with Network-Attached Secure Disks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems 1997*, June 1997. To Appear.

[7] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Erik Riedel, David Rochberg, and Jim Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, March 1997.

[8] Li Gong. A Secure Identity-Based Capability System. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–64, May 1989.

[9] National Storage Industry Consortium NASD Working Group. http://www.hpl.hp.com/SSP/NASD/.

[10] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 1993.

[11] Mark Hayter and Derek McAuley. The Desk Area Network. *ACM Transactions on Operating Systems*, pages 14–21, October 1991.

[12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[13] Henry M. Levy. *Capability-based Computer Systems*. Digital Press, Bedford, Mass., 1984.

[14] Rodney Van Meter, Steve Hotz, and Gregory Finn. Derived Virtual Devices: A Secure Distributed File System Mechanism. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, September 1996.

[15] D. Mills. Network Time Protocol (v3). Technical Report RFC 1305, Network Working Group, April 1992.

[16] D. Mills. Simple Network Time Protocol (SNTP). Technical Report RFC 1768, Network Working Group, March 1995.

[17] B. Clifford Neuman. Proxy-Based Authorization and Accounting for Distributed Systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[18] U. S. National Institute of Standards and Technology. Federal Information Processing Standards Publication 186: Digital Signature Standard, May 1994.

[19] Hawaii International Conference on System Science 30 Taskforce on Network Storage Architecture. http://www.pdl.cs.cmu.edu/NASD/HICSS.html.

[20] E. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, New York, New York, 1983.

[21] Michael Rabin. Fingerprinting by Random Polynomials. Technical Report TR-81-15, Center for Research in Computing Technology, Aiken Laboratory, Harvard University, May 1981.

[22] R. Rivest. The MD5 Message-Digest Algorithm. Technical Report RFC 1321, IETF Network Working Group, April 1992.

[23] R Sandberg. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119 – 130. USENIX, June 1985.

[24] Robert D. Sansom, Daniel P. Julin, and Richard F. Rashid. Extending a Capability Based System into a Network Environment. In *SIGCOMM '86: Futures in Communications*, August 1986.

[25] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using Sparse Capabilities in a Distributed System. In *Proceedings of the Sixth International Conference on Distributed Commputing*, pages 558–563, 1986.

[26] Joseph Touch. Performance Analysis of MD5. In *Proceedings of SIGCOMM 1995*, pages 77–86, 1995.

[27] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. Nort Holland, 1979.

[28] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[29] Bennet Yee and Doug Tygar. Secure Coprocessors in Electronic Commerce Applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.