UNIVERSITY of CALIFORNIA
SANTA CRUZ

USING NETWORK ATTACHED STORAGE IN A SECURED DISTRIBUTED FILE SYSTEM

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Benjamin Clay Reed

June 2000

The dissertation of Benjamin Clay Reed is approved:

Prof. Darrell D. E. Long, Chair

Prof. Tara M. Madhyastha

Dr. Cynthia Dwork

Dr. Ran Canetti

Dean of Graduate Studies

Copyright © by

Benjamin Clay Reed

2000

Contents

List of Figures vi

List of Tables vii

Abstract viii

Acknowledgments x

1 Introduction 1
- 1.1 Distributed File Systems . . . 3
- 1.2 Related Work . . . 4
  - 1.2.1 Distributed File Systems . . . 4
  - 1.2.2 Methods of Authentication . . . 16
- 1.3 Overview of the Thesis . . . 25

2 The SCARED Object Model 27
- 2.1 Object Abstraction . . . 28
  - 2.1.1 Basic Semantics . . . 29
  - 2.1.2 Data Object Semantics . . . 30
  - 2.1.3 Metadata Object Semantics . . . 31
- 2.2 Block Allocation . . . 33
- 2.3 Cache Management . . . 34
- 2.4 Access Control . . . 36
- 2.5 Summary . . . 38

3 Deriving Keys for Authentication 39
- 3.1 Distributed SCARED Environment . . . 41
- 3.2 Key Distribution Without Key Exchange . . . 42
- 3.3 Key Types . . . 43
  - 3.3.1 Generating Capability Keys . . . 44
  - 3.3.2 Generating Identity Keys . . . 45
  - 3.3.3 Combining Keys . . . 46
- 3.4 Revocation . . . 48
  - 3.4.1 Key Expiration . . . 48

3.4.2 Capability Key Revocation 49
3.4.3 Identity Key Revocation 49
3.5 Security Analysis 50
3.6 Summary 55

4 An Authenticated Message protocol for SCARED 57
4.1 Integrity and Identity Guarantees 59
4.2 Freshness Guarantees 61
4.2.1 Verifying Freshness using Counters 63
4.2.2 Verifying Freshness using Timers 63
4.3 The Request/Response Protocol 64
4.3.1 The Request Protocol 64
4.3.2 Response Protocol 65
4.3.3 Asynchronous Responses 66
4.4 Encryption 67
4.5 Analysis of Message Protocol 68
4.5.1 Exchanging the Freshness Guarantee 69
4.5.2 The Generic Message Protocol 70
4.6 Summary 74

5 Using SCARED in a Distributed File System 75
5.1 Brave File System Layout 76
5.2 Brave Semantics 79
5.2.1 File Semantics 79
5.2.2 Directory Semantics 80
5.3 Brave Operations 80
5.3.1 Creation 81
5.3.2 Deletion 82
5.3.3 File System Checks 83
5.4 Conclusion 84

6 Implementing SCARED and Brave 85
6.1 UNIX file systems 86
6.1.1 I-nodes 86
6.1.2 Directories 86
6.1.3 Virtual File Systems 87
6.2 Integrating Brave and SCARED into the VFS 88
6.2.1 Allocation Management 90
6.3 Implementing SCARED 91
6.4 Summary 92

7 Conclusions 93
7.1 Contributions 93
7.1.1 Comparison to Related Work 93
7.1.2 Specific advantages of SCARED and Brave 95
7.2 Future Work 97

7.2.1 Caching 97
7.2.2 Locking 98
7.2.3 Striping and Mirroring 98
7.2.4 Allocation and Load Balancing 99

A Key Data Encoding **101**
A.1 Identity Attribute 102
A.2 Capability Attributes 103
A.3 Key Information Attributes 104
A.4 Key Data Evaluation 105

B Pseudo-Random Functions **106**

Bibliography **108**

List of Figures

1.1 The data and metadata accesses in NFS and CIFS. 5
1.2 The data and metadata accesses in AFS. 9
1.3 Client access using data and metadata servers. 11
1.4 Client access using NASD. 13

2.1 The structure of a file object. 30
2.2 The structure of a metadata object. 31

3.1 The administrator, the storage device, and the client are the three roles in SCARED. The key derivation scheme allows the administrator to generate access keys for the clients. 41
3.2 The administrator shares a key, K, with the storage device which is used to generate keys to be given to the clients. In this example the messages must be exchanged over secure channels. 45
3.3 Mixing identity and capability keys to enable printer access to a data object. 47

5.1 Brave directory entry layout. 76
5.2 An example directory structure stored in a meta data object. 78

6.1 Brave integration into the Linux VFS. 88

List of Tables

A.1 SCARED attribute types for key data. 102
A.2 Permission masks for the permission capability attribute. 103

Network Attached Storage in a Secured Distributed File System

Benjamin Reed

Abstract

Distributed file servers are becoming an important part of the network infrastructure. The increased capacity of disk drives has increased the amount of storage managed by the file server. The number of network clients have increased, as well as the bandwidth and connectivity between the clients and servers. The file server is a bottleneck in the access path between the network client and the data on the disks. To alleviate this bottleneck it has been proposed to directly attach disks to the network, thereby increasing the aggregate network bandwidth to the data and relieving the file server. Attaching disks to the network brings security problems that do not exist when the disk is only attached to the file server.

Simply applying existing authentication protocols to network attached storage is not sufficient because of their administrative and computational requirements. We review some of the common means of authentication in use today and their weaknesses when applied to network attached storage.

To address these authentication weaknesses, we present an authentication protocol to provide strong authentication guarantees to network attached storage. This protocol avoids the infrastructure and computational overhead of other protocols while still providing strong identity, integrity, and freshness guarantees.

To enable the protocol we introduce an object model to permit the correct level of access control to the data stored on the network storage devices. Additional advantages to using an object interface as opposed to a block interface are discussed.

We describe a completely distributed file system, which we implemented for Linux, that

takes advantage of the authentication protocol and object model. The file system exhibits scalability, manageability, and security features missing in most contemporary file systems. It also illustrates how adding simple object semantics to network storage devices can remove the need for a file server without sacrificing security.

Acknowledgments

I must first thank Mom for teaching me to love to learn and Dad for his guidance and example. My secondary education in Fairfield, Ohio, provided a foundation on which to build. Miami University taught me how fun learning can be. DePaul University showed me the wonders of Computer Science, and the University of California, Santa Cruz, taught me to explore. Unfortunately, space doesn't permit me to acknowledge all the wonderful teachers and advisors I have had at these institutions. At Santa Cruz, in particular, I enjoyed and learned immensely from every instructor I had and worked for. If there were not so many, I would be able to acknowledge them individually.

Special acknowledgement goes to my advisor, Darrell Long, without whose patience and guidance I would have never been able to finish (or start, for that matter). It was his comments in his Computer Security class that inspired this work. It was the freedom I had at IBM's Almaden Research Center that let me finish this particular work. I must especially thank my managers, Steve Welch and Norm Pass, who refused to let me not get my degree.

The advancement process of the doctoral program requires special efforts on the part of the advancement committee and the defense committee. I am truly grateful to Alexandre Brandwajn, Cynthia Dwork, Darrell Long, and Anujan Varma, who were on my advancement committee. As well as Darrell Long, Tara Madhyastha, Cynthia Dwork, and Ran Canetti for their work on my defense committee.

Cynthia and Ran have taught me and guided me in the security aspects of this thesis. I certainly would not have been able to get it right without their help. Ed Chron, Randal Burns, and Darrell Long helped me with the systems part of this work. Ed's support and encouragement, in particular, helped me continue this work. Ben Gertzfield did the initial Linux VFS work and helped us flush out the design of the Brave client.

Obviously, every good idea is built on others and the influence of my colleges and the body of published work in this area helped me pull the needed pieces together. Most of all, I must also acknowledge God's efforts in getting a couple of good ideas through my thick skull, and the unwavering support and encouragement of my wife, Carolina.

This dissertation is dedicated to my wife and children.

Chapter 1

Introduction

The need to access anything from anywhere has emphasized the role of distributed file servers in computing. Distributed file systems provide local file system semantics when accessing remote storage. This allows network clients to incorporate the remote storage into the local file system. File semantics are well understood by users and applications, making distributed file servers a convenient tool to use in developing distributed applications.

As the role played by distributed file systems expands, some shortcomings of their design become increasingly evident. Faster clients, high bandwidth connections, and larger drive capacities increase the demand on file servers. Although it would seem that network file server performance would be limited by the I/O capacity of the system storage devices, in actuality, with sufficient I/O bandwidth, file servers frequently become CPU bound. Riedel and Gibson showed that even with low overall CPU utilization burst loads were sufficiently intense to over-utilize the CPU [47]. By allowing direct access to storage devices by the clients they were able to reduce the workload of the file servers. This kind of direct access also requires a supporting authentication mechanism to prevent malicious clients from making unauthorized changes to the storage and, consequently, the

file system. While this kind of access control becomes more apparent when clients can directly access the storage devices, even classical distributed file systems are frequently lacking in this area.

Applications that rely on distributed file systems should not be compromised by security weaknesses of the file systems on which they are built. Data stored on distributed file systems frequently need to be protected from unauthorized access or eavesdropping. The administrators of the distributed file servers control access to the servers and, consequently, who has access to the data on their storage devices. Encryption can be used to preserve the confidentiality of the data in these situations, but in practice users must encrypt outside of the file system to achieve this kind of confidentiality. Contemporary distributed file systems are only beginning to address these issues.

The authenticated network attached disks we present address these problems by providing an architecture based on one-way hash functions providing for mutual authentication of the network disks and the clients. This architecture obviates the need for more performance intensive authentication methods such as public-key encryption and Kerberos [40]. The authentication infrastructure required is very small and flexible, allowing it to fit into more complex systems.

Finally, since encryption is not required to support authentication, a variety of legal issues can be avoided. Domestic encryption is restricted in some countries [9], and others restrict export of encryption [55, 56]. These restrictions can be avoided by the network disks allowing the same disks to be used worldwide.

We review the components of distributed file systems in §1.1. In §1.2 an overview of the types of contemporary file systems is presented to show the context in which this work was done, as well as an overview of the authentication methods used in those file systems. An overview of the rest of this thesis is presented in §1.3.

1.1 Distributed File Systems

In general, a distributed file system has four components: clients, file servers, authentication servers, and data stores. Client machines access files on behalf of users and applications. Users and their applications have credentials that are used to identify themselves to an authentication server, or even directly to a file server. Some examples of file systems, which will be reviewed later, that separate the file server and the data stores are Swift [32] and Zebra [22]. Andrew File System (AFS) [24] is an example of a file system that separates the authentication and file server components. The more contemporary file servers, such as Network File System (NFS) [53] and Common Internet File System (CIFS) [41], do the file serving, authentication, and storage at the same server.

If an authentication server is present, the client authenticates the user to the authentication server in the form of a password, token, or other authentication method. The authentication server gives the client new tokens that are used to access the file server. These tokens may grant access to specific files on the file server or may simply authenticate the identity of the user.

In classical distributed file systems, all accesses to the data store are through the file server. The file server verifies the accessibility of the data before carrying out the request from a client on the data store. The data store is usually locally attached to the file server. Since the local storage is only attached to the file server, it can simply carry out the requests of the file server without having to authenticate or check access permissions.

Storing data on a network is often accompanied by the sharing of data between users. For sharing to occur, users need to be able to transfer rights to other users. Assignment of a user's rights or of a subset of those rights to another user should be possible. On some distributed file systems, a user can give access to specific files. On others, the granularity of sharing is at the directory level.

Finally, some only allow users to grant access to entire sub trees.

1.2 Related Work

This section presents some key work in the area of distributed file systems to illustrate the differences in their aspects. We start by presenting some of the popular server based file systems, followed by file systems that distribute the work across multiple servers, and then completely distributed file systems. After presenting these file systems, we will present work in the area of authentication, followed by some important examples of the application of these authentication methods in distributed file systems, beginning with the weakest forms of authentication.

1.2.1 Distributed File Systems

While most distributed file systems share the common goal of extending local file system semantics to network storage, the approaches differ greatly. To illustrate these approaches we begin by presenting two of the most popular network file systems: NFS and CIFS. We then present AFS and DFS. They allow the file system to be spread across multiple servers. We also present a group of file systems that manage the file system metadata at file servers and store the data on dedicated data servers. This idea has been applied to NFS and AFS by the NASD project at Carnegie Mellon University. Finally two serverless distributed file systems are presented.

NFS

NFS [53] was developed by Sun Microsystems to provide transparent remote access to files. It uses a Remote Procedure Call / Extended Data Representation (RPC/XDR) interface to make it portable across operating environments. The file system consists of stateless file servers and

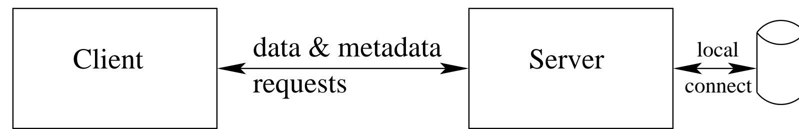

Figure 1.1: The data and metadata accesses in NFS and CIFS.

file system clients. NFS is more a file sharing protocol than a file system. The NFS protocol implies mostly UNIX semantics to the files, and the files themselves are usually stored in a local file system on the file server. The file server exposes sub trees of the local file systems to the network.

Since the files are being shared from a local file system, file systems accessed by NFS clients do not span servers. CIFS, described in the next section, shares this limitation. Figure 1.1 illustrates the file system access in these server based network file systems.

When a client connects to a file server the client first uses the mount protocol to get a handle to the root of the sub tree that will be accessed. The mount protocol server runs on a privileged port. Once the handle is obtained, the clients communicate with the file server running on a non-privileged port to request file and directory data. Version 2 of the protocol restricted the maximum transfer size per request to 8192 bytes. Version 3 [8] of the protocol removed this limit which allows for better performance. Client caching is not specified in the protocol, but in practice NFS clients cache file data for 5 seconds and directory data for 30 seconds. Writes are committed to disk when received by the server. Version 3 added a write commit protocol to allow multiple writes before actually committing to disk.

The most common form of authentication in NFS is network based. Only the server authenticates the client. The client does not authenticate the server. As mentioned in section 1.2.2, network based authentication is subject to a number of attacks and tools exist to exploit them. Other proposals exist for using DES with public key encryption and Kerberos, but they have yet to gain

popularity.

NFS is stateless to make it resistant to server failures. Because it is stateless, it loses the open and close semantics of files. Open and close semantics can be used for efficient cache management both on server and client. Caching reads is very important since reads are the overwhelming majority of client operations. The simple time based caching limits the effectiveness of the cache and does not assure cache consistency.

CIFS

The Common Internet File System (CIFS) [41] is a stateful file sharing protocol that has evolved from a file and print sharing protocol for personal computers on a local area network. A CIFS server is able to share files, printers, and FIFO (or named pipes) with CIFS clients using Server Message Blocks (SMB). Communication between the client and server takes place via a request SMB and a response SMB. With one exception, the client always makes a request to the server not *vice-versa*. SMB's are encapsulated in a NetBIOS packet and transported over a reliable transport such as TCP.

CIFS uses mandatory locking that is enforced by the CIFS server. Files may be opened with file level locks to provide read, write, and exclusive locks to a file. Byte locking can also be used to lock ranges of a file for read or write access. Variations of the read and write SMB's are available to perform a simultaneous read and lock on a byte range, as well as a simultaneous write and unlock. In the case of the level locks and byte range locks, the locks will be released when the client explicitly releases the locks or closes the file. The CIFS server will not revoke the locks held by a client. The later versions of CIFS added a temporary locking called opportunistic locking that can be requested by the client when a file is opened. If opportunistic locking is requested,

the client need not write changes to the server or request locks until the file is to be closed or the server revokes the opportunistic lock. If the server must revoke an opportunistic lock, the server sends an SMB to the client revoking the lock. The client replies to the request after flushing any data to be written and requesting locks for ranges that were requested while the opportunistic lock was held. The specification for the CIFS protocol makes very little mention of caching. Caching is only mentioned in relation to opportunistic locking. Cache consistency can be maintained using the CIFS locking mechanism. However, since the server cannot revoke locks held by the client the caches must be flushed to the server quickly and the locks released to avoid impacting other clients waiting on the lock.

CIFS supports two security modes: share-level and user-level. In both cases the client and server are both in possession of a shared secret (*i.e.* the password). Security is enforced when a session is initiated with a resource. In the case of share-level security, access to the resource is restricted using a password. Once access is gained to a resource using share-level security, the same level of access is used for all files and sub directories in the resource. User-level security, on the other hand, requires a userid and password to access the resource. Once the resource is accessed the level of access can vary on an individual file or directory basis.

To prevent eavesdropping of passwords when initially accessing resources of a CIFS server, encryption can be used while authenticating. DES is used as the encryption algorithm. The variable names (P_i) and terms used to describe the validation protocol are taken from the standard [41]. When the client initially connects to the server and negotiates the protocol level that will be used, the server sends back a crypt key which is computed by encrypting a string composed of eight question marks with a seven byte string, which is usually a combination of the time and a counter. The client begins encrypting the password by calculating P_{16}, which is a string composed of eight

question marks encrypted with P_{14}. P_{14} is the user's password padded with spaces if necessary to form a 14 byte string. P_{21} is P_{16} with five null bytes appended. Finally, the encrypted password is the result of encrypting the crypt key with P_{21}. Both the client and server perform the calculation and the server validates the result sent by the client with the result of its own calculation.

If authentication is done using the above algorithm, the password is protected from eavesdropping and the use of the crypt key seems to protect the result from being replayed to gain access. The authentication is only done on initial connection to the resource. The connection oriented protocol is relied upon to maintain the integrity of the rest of the session. As was pointed out earlier, TCP cannot be relied upon to provide this kind of integrity. Packets can be manipulated to and from the server in order to fool the client and server; or the session itself could be taken over after the connection is made, giving the attacker access to the resource as if she were the user that connected to the resource.

Dedicated Server

General purpose operating systems provide many functions and features that are superfluous to a system whose sole function is to act as a file server. Some of these functions include graphical user interfaces, multitasking, and application programming support. Operating systems have been created to expressly support the file serving function. An example of such an operating system is Network Appliance's dedicated file server. Their customized operating system is designed specifically for processing network requests, and includes a Write Anywhere File Layout (WAFL) [23] that is optimized for file serving. The result is a file server with improved performance when compared to a file server hosted by a general purpose operating system. Several existing file sharing protocols are supported including CIFS, the *de facto* Windows file sharing protocol, and NFS, the

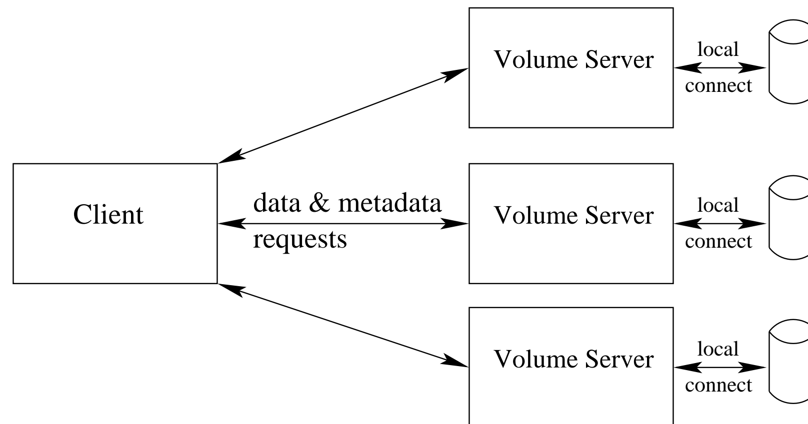

Figure 1.2: The data and metadata accesses in AFS.

de facto UNIX file sharing protocol. This allows the server to communicate with many existing network clients.

AFS/DFS

Andrew File System (AFS) [24] is a stateful distributed file system. It has a much more complicated infrastructure than NFS because it actually is a file system. AFS presents the network clients with the appearance of a single name space. The first level of the name space exposes the available AFS cells. Each cell has a subtree in the AFS network complete with its own domain of authentication. These cells can have many volume servers. The volume servers contain the actual file system data. Figure 1.2 illustrates the access methods of the clients. The figure shows that the metadata and data requests are passed to volume server. Each volume server contains a subtree of the file system.

Its performance advantage over other distributed file systems is a result of its caching the entire file locally when it is opened. Cached copies of files are made consistent with the file on the file server when the file is closed. Benchmarks show that local caching of files reduce the load on

the file server allowing it to serve additional clients. AFS uses a callback mechanism to keep copies of files that are in the cache and no longer opened consistent with the originals on the file server. The performance improvements in AFS are not universal. If a file is not cached, a file open may take longer than with NFS. File closes may take longer than using NFS, since the data must be flushed to the server if the file has changed.

Authentication in AFS is done using Kerberos [40, 29]. Directories have access lists which allow accesses based on user's Kerberos identifier. When a user wishes to access AFS, she first obtains a Ticket Granting Ticket (TGT) from the authentication server. This ticket is time limited and allows the user to request additional tickets to communicate with the AFS file servers. Kerberos tickets are only used for authentication; no encryption is done on the actual file data. Furthermore, even though all the network packets sent between the clients and file servers are authenticated using Kerberos, the packets themselves are not actually integrity protected. For performance reasons, only the packet header has integrity guarantees, so modifications to the payload of the packets are undetected. Kerberos relies on synchronized clocks to prevent replays. If clocks are not loosely synchronized, the authentication services will not work.

Decorum File System (DFS) [27] is a follow-on to AFS. It improved caching by allowing parts of the file to be cached instead of requiring the whole file to be cached. This was necessary to allow for files to be opened that would be too large to cache in their entirety. The improved file caching also included cache consistency call-backs with finer levels of granularity. Files in DFS are always kept consistent. When a file is opened, the client obtains a token for the piece of the file that is kept in its local cache. If the token is for writing and another client requests that piece of the file, the server will revoke the token, cause the client to flush any changes to the server, and release the token. After the first client releases its token, the second client will be granted a token and make

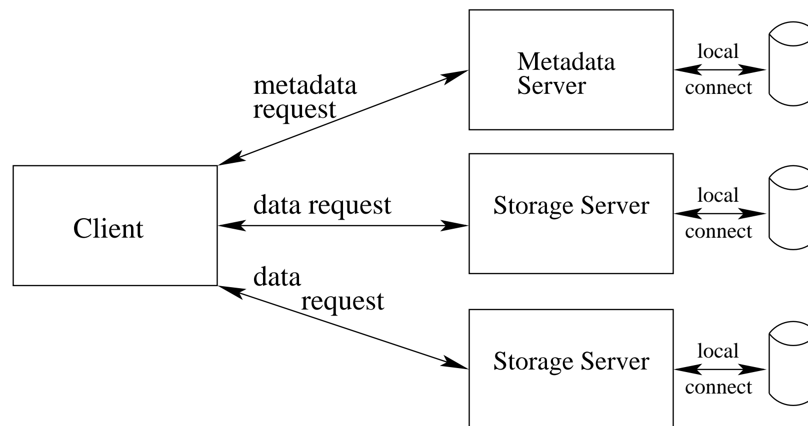

Figure 1.3: Client access using data and metadata servers.

changes to, or read, that piece of the file.

The latest version of DFS [15] also enhances security. All packets are integrity protected by default and there is an option for encrypting the packets between the client and server. The files themselves are still stored in plain text on the file servers.

Storage Servers

Figure 1.3 illustrates how metadata and data can be managed separately by separate servers. The simplest example of a file server making this kind of separation is the Bullet [59] file system. It had directory servers that handled naming and access control, and Bullet servers that store file data in immutable data objects. Access to the data objects on the Bullet servers was done with object numbers to identify a data object and capabilities to allow client access to a given object. The object numbers and capabilities were distributed by the directory servers. The greatest limitation of the Bullet file system was that file data objects must be read and written in their entirety. Given that an analysis of the file system traffic at the time [42] showed that 75% of the files are accessed in their entirety, the authors did not view this limitation as that great.

The Swift file system [32] also used objects at the storage servers, but they were used to stripe the file data across objects on different storage servers. It also did not require that files be accessed in their entirety. The authors found that this striping boosted the aggregate bandwidth and processing available to serve files even when individual files were being accessed.

Because individual files were striped across the storage servers, files that were smaller than the stripe stride multiplied by the number of storage servers were not able to fully obtain the benefits of striping. Zebra [22] combined the ideas Swift [32] and log-structured file systems (LFS) [51] and RAID [44] to produce a file system that stripes the file system data across the storage servers. Instead of storing file data in objects and the storage servers, each client writes update logs to its own stripe fragments and the network storage. The clients use the file manager (which manages the metadata) to manage the mapping between the file system name space and the location of the file data in the stripe fragments.

Network Attached Storage Device

Dedicated servers provide an impressive increase in performance. However, the scalability of the server is restricted by factors such as the processing power of the CPU, the speed of the system bus, network interface, disk interface, and the disks themselves. This is because the file server must be involved with all transactions between the client and the disks. Figure 1.4 illustrates one solution to this problem. The disks are directly attached to the network and allow the file server to marshal client requests to the appropriate disks. When disks are network attached, the file server is able to manage significantly more storage and the aggregate network bandwidth increases dramatically.

The convergence of network and I/O connection technologies inspire investigations into

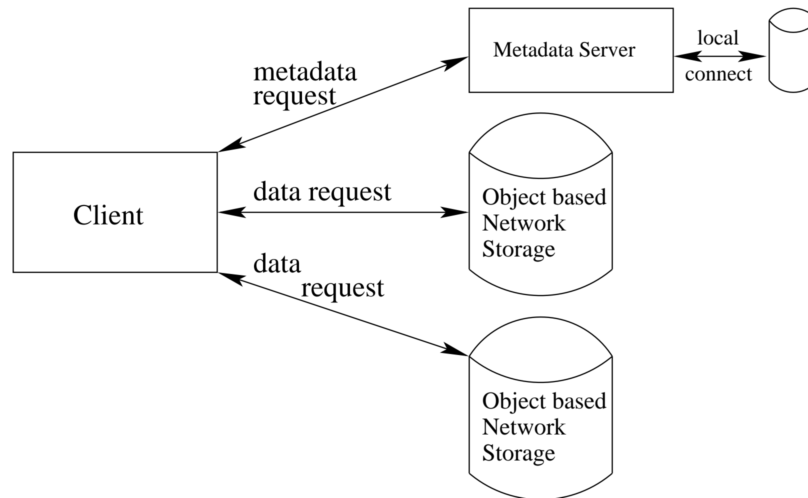

Figure 1.4: Client access using NASD.

the use of the network as the I/O bus [52, 26]. For example, Network-Attached Secure Disks (NASD) [17] have been made to operate with Network File System (NFS) and Andrew File System (AFS).

In file systems built with NASD, the file server provides only file system meta-data while the actual file data are provided by the network attached disks. When files are accessed by network clients, the requests must be authenticated and permissions checked before the access is allowed. The file servers generally do this kind of checking. However, if clients are allowed to directly access the disks, the disks must also be able to verify the authority of the client's access to the data.

NASD uses time limited capabilities to authorize clients to perform specific actions on the disk. The disk has shared secrets with the file manager that are used to create capability keys. NASD initially used DES and secure communication between the file server and the NASD to coordinate the capabilities. Later [18], capabilities were generated using the shared secrets at the file manager in such a way that the NASD could verify a capability that was sent by a client was created by the

file manager. This optimization eliminated a lot of key exchange traffic between the file manager and the NASD.

When a client wishes to access data on the disk, the client gets a capability from the file server to present to the network disk. The client then presents the capability with a request to the disk, at which point the disk verifies that the capability allows the requested action before actually carrying out the action.

Benchmarks show that by allowing clients direct access to the network attached disk, the processor utilization of the file server decreases dramatically. This means that the file server can handle more storage, but does not eliminate the file server altogether. Therefore, the scalability of the file server limits overall growth of the file system.

Serverless

The serverless file system (xFS) [11] was developed in conjunction with the Network of Workstations (NOW) project at the University of California, Berkeley. The file system consists of a network of trusted workstations that cooperate to provide the functionality normally provided by a network server. The result is a file system that has no central repository of files. Instead, the file system data and meta-data is spread among the network of trusted workstations.

By spreading the file system data and meta-data across multiple machines, the aggregated resources are much greater than what would be found on a normal file server. These resources include cache size, network bandwidth, and processing power. Striping and logging is also used to boost performance.

While dramatic performance improvements are seen when the serverless file system is compared to a more centralized file system, these performance improvements come at a price: re-

duced security. The client and manager kernels are trusted to protect the file system from malicious access. Thus, the serverless file system is designed to run in a uniform security environment. This kind of environment can be found in NOW and in a network where all machines are administered and trusted equally.

To allow access to the serverless file system by untrusted clients, an NFS gateway is used. The gateway is trusted and serves as a firewall between the trusted network and the untrusted clients. NFS clients access the gateway as an NFS server. The gateway then makes requests on the part of the untrusted client. The gateway has the potential to become a bottleneck and adds extra traffic and processing to the requests. However, performance improvements are still obtained via the cooperative caching and log-based striping.

JetFile

JetFile [21] is similar to the serverless file system in that file system data is spread across the clients. However, JetFile uses storage servers to act as a repository of files for backup and availability reasons. The key design point of JetFile is their use of multicasts as the communications medium. Multicast is used to ensure cache consistency amount all the clients accessing a file system object.

Each file system object is given a FileID. Associated with a FileID is a multicast address. Scalable Reliable Multicast (SRM) [16] is used to locate a FileID, and unicast is used to actually retrieve the data from that location. The client that has the FileID will answer the SRM and serve the data to the requesting client. Writes are done locally and do not need to be written back to a storage server, since the client doing the writes will serve the data to other clients by answering SRMs for the FileID with itself as the location. Before a modified file is removed from the local

cache, the client will update the storage server with the changes.

Because writes and reads do not have to go back to a storage server and many replicas of a file can exist at different clients, it is important to have a way to serialize updates to a file. Serialization is done using a version server that generates version numbers for files that are updated by clients. The version requests are also multicast, so other clients can mark their cached files as changing. The version server also periodically sends out a table of current file versions via multicast to fix up any clients whose version table has become out of sync.

1.2.2 Methods of Authentication

Users of a distributed file system have the ability to control who has access and the kind of operations that can be performed on their files. These restrictions are normally enforced by the file server. However, before a file server can allow an action to be taken on a piece of data, it must first discover the identity of the requester of the action. The file server requires the user to authenticate with it before deciding whether the request is to be allowed.

In some cases authentication is done by simply retrieving the user identifier from the request; in others, cryptography is used to give a stronger way of validating identity.

Authentication is not a concept that applies only to clients. Clients may also wish to validate that the response to a request came from the server from whom they believe it came. The most common distributed file systems in use today only authenticate clients.

Network Based

Every device connected to a TCP/IP network has an IP address associated with it. This IP address is unique to the network. When two devices communicate with each other, each message

sent includes the source IP address and the destination IP address. It is convenient to authenticate the source of a message by checking the source IP address. Currently, IP addresses are four bytes, usually written as four numbers separated by dots. Since numbers are hard to work with, IP addresses are mapped to domain names by DNS [36]. This mapping allows the use of more descriptive alphanumeric names for IP devices. Many applications that authenticate machines based upon network addresses allow the use of domain names in the access lists.

Usually network applications authenticate users, not machines; so further information about the user sending the message is needed from the remote machine. UNIX systems have a userid associated with each user. This userid is a 16-bit number which is mapped to an alphanumeric user name generally done via an **/etc/passwd** file or Network Information System (NIS). When messages are sent to servers, the client usually includes information about the user that is making the request. The server is more likely to trust that the message contains a valid user name if the message is coming from a privileged port. UNIX systems allow only the super user (root) to access privileged ports (numbered below 1024).

In theory, the use of network address authentication of machines and privileged ports provide a good means of client and server authentication. However, in practice a number of attacks [4] can compromise the security of both the client and server. Early on, attacks on routing protocols allowed one device to spoof (or act like) another device by taking on the network address of the device and redirecting network routes. Spoofing can also be done by returning fraudulent data to requests to resolve domain names from IP addresses. Also new PC operating systems, such as OS/2 and Windows, do not have a concept of privileged ports; thus making it difficult to trust any user information sent by the machine.

Password Based

CIFS, described in §1.2.1, uses password based authentication. This type of authentication is also common in file transfer protocols such as FTP and HTTP. In its simplest form, a client initiates a connection with a server and sends a user identifier and a password that corresponds to that user on the server. The password is a shared secret between the user and the client.

Of course if the password is sent in the clear, it is exposed to a network eavesdropper. To avoid exposing the password it may be exchanged via a challenge-response type of protocol. In theory, the challenge-response could be used to validate that both the client and server are in possession of the key; but, frequently, only the server issues the challenge.

Even if the password is not sent over the network in clear text, it is still susceptible to man-in-the-middle and session takeover attacks. In the man-in-the-middle attack, the attacker relays traffic between the client and server until they have finished validating each other, at which point the attacker starts making requests directly to the server acting as if it were the client. In session takeover, the attacker takes on the network identity of the client after the client has authenticated itself to the server. Both attacks take advantage of the fact that authentication is only done at the beginning of the session.

Kerberos

Kerberos [40, 29] bypasses problems with network based authentication by not trusting the network. It is an example of an authentication method that uses a trusted third-party and symmetric encryption. The Kerberos protocol adds time stamps, a ticket granting server, and another approach to cross-realm authentication to the Needham and Schroeder authentication protocol [38]. There are four entities in the Kerberos authentication protocol: the client, the server, the Authenti-

cation Server (AS), and the Ticket Granting Server (TGS).

The client initiates the communication between client and server. A Kerberos ticket is used by the client to authenticate itself to the server. The ticket contains a certificate issued by an AS or TGS. The certificate includes a random session key, the identity of the client, and an expiration time. The certificate is encrypted with the secret key of the server with whom the client will communicate. The client requests tickets from an AS or a TGS using a shared secret key.

The authentication server has a database of secret keys used by clients, servers, and ticket granting servers. When a client wishes to establish the identity of the user, on whose part it is acting, to a server, the client requests the password from the user and then requests a ticket from the AS to communicate with the server on behalf of the user. The AS responds with a ticket encrypted with the secret key of the server and the random session key included in the ticket encrypted with the password of the user. Tickets received from an authentication server are called Ticket Granting Tickets (TGT) because they are used to obtain other tickets from a TGS. A TGT is used to allow single sign-on. The user should only have to give her password once instead of every time she needs a ticket. Keeping the password in memory is dangerous, since an attacker that obtains the password could impersonate the user until the password is changed. By storing a TGT the attacker could only use the TGT until expired (usually on the order of eight hours).

Once the client has a TGT, it can be used to obtain tickets from a TGS to authenticate the client to other servers. When a client wishes to establish the identity of its user to a server, the client presents the TGT and the identity of the server to the TGS. The TGS returns a ticket to the client encrypted with the secret key of the server and the random session key of the ticket encrypted with the random session key in the TGT. The client presents the ticket to the server and then they exchange cipher text encrypted with the random session key of the ticket to mutually authenticate

themselves.

Implementations of Kerberos use DES, which can cause problems when trying to export applications which use Kerberos. Although DES keys are difficult to break, humans need keys that they can remember. Keys chosen by users are often subject to dictionary attacks. In addition the infrastructure which Kerberos requires can be difficult to setup and maintain. The expiration time of the tickets require synchronized clocks which may be difficult to achieve with network attached storage. If clocks get out of synchronization, replays and broken keys can be used to attack servers. These limitations with others have been detailed by Bellovin and Merritt [3].

Public Key

In 1976, Whitfield Diffie and Martin Hellman published a paper [13] that proposed public-key cryptography. Ralph Merkle had proposed the first implementation of a public-key cryptosystem two years earlier in a term paper [35]. A public-key cryptosystem consists of two keys: a public-key and a private-key. The private-key is computationally difficult to derive from the public-key. Users of the cryptosystem distribute their public-key keeping the corresponding private key secret. Messages are sent to a user by encrypting the message with the user's public-key. The message can then only be decrypted by the user in possession of private-key.

The Diffie Hellman algorithm is a key exchange algorithm based on public-key encryption. The algorithm gets its security from the difficulty of calculating discrete logarithms compared to the ease of doing exponentiation. In the key exchange the two parties have a common n and g such that g is primitive with respect to n. Both n and g may be public. The first party picks a random number x, and the second party a random number y. Where x and y are the private keys. The two parties then exchange g^x and g^y. These two results constitute the public keys. Both parties

now have a shared secret: g^{xy}, which they can use as a shared symmetric encryption key. One attack that Diffie Hellman is subject to is the man-in-the-middle attack. In man-in-the-middle, the attacker does a key exchange with each party as if it were the other. Since there is no authentication in the algorithm neither party knows the identity of the other.

Authentication can be done with public-key algorithms through the use of digital signatures. In these algorithms the private key is applied to a function along with the message to be signed. The result is the digital signature. A recipient of a message can then apply another function with the message and the public key and verify that the result matches the digital signature. There have been many digital signature algorithms devised. The most popular are Digital Signature Algorithm (DSA) and Rivest Shamir and Adleman (RSA).

Digital Signature Algorithm is part of the Digital Signature Standard [57]. This standard was introduced in 1991 to provide message authentication and integrity. It does not provide encryption. The algorithm has three public parameters: p, a prime number, q, a prime factor of $p-1$, and $g = h^{(p-1)/q} \bmod p$, where h less than $p-1$ and $h^{(p-1)/q} \bmod p > 1$. The private key is x which is a number less than q. The public key is $y = g^x \bmod p$. A message, m, can be signed by computing $r = (g^k \bmod p) \bmod q)$ and $s = (k^{-1}(H(m) + xr)) \bmod q$. The function H is a one-wash hash function (DSA requires the use of Secure Hash Algorithm [58]), and r and s constitute the signature of m. Anyone can verify the signature by calculating $w = s^{-1} \bmod q$, $u_1 = (H(m) \times w) \bmod q$, $u_2 = (rw) \bmod q$, and $v = ((g^{u_1} \times y^{u_2}) \bmod p) \bmod q$, then verifying that $v = r$.

RSA [50] predates DSA and is much simpler. RSA also provides encryption as well as authentication. The public key consists of $n = pq$, where p and q are prime, and e that is relatively prime to $(p-1)(q-1)$. The private key is $de \equiv 1 \bmod (p-1)(q-1)$. After generating d, p and q are discarded and never revealed. RSA can be used to sign a message, m, by signing its hash:

$h = H(m)$. The digital signature is given by $s = h^d \bmod n$. Anyone can validate a signature by computing $v = s^e \bmod n$ and verifying that $v = h$.

One problem with public-key cryptosystems is key distribution. Two parties can mutually authenticate themselves only if they know the signing key of the other party. Managing keys becomes an intractable problem if everyone's public-key must be distributed to every other party before communication takes place. This problem is addressed by the use of Certificate Authorities (CA). Everyone is required to have the public-key of the CA. In addition everyone must have their public-key signed by the public-key of the CA. This is generally done by getting a message, called a certificate, with the party's identity and public-key in it signed. When one party sends a message to the other, it signs the message with its public-key and sends the message with the signature and its certificate. The other party can verify the owner of the public-key via the certificate and the authenticity of the message by using the public-key contained in the certificate.

One of the problems of certificate based authentication is revoking a certificate. A certificate would need to be revoked if the private-key was lost, for example. Certificate revocation is dealt with in two ways: expiration dates, and Certificate Revocation Lists (CRL). Each certificate contains an expiration date that limits the amount of time that a compromised key can be used. A CRL contains a list of certificates that are no longer valid. This list is signed by the CA and distributed periodically. Because each certificate has an expiration time, the list will not grow indefinitely.

The two major drawbacks to digital signatures are key size and speed. Both DSA and RSA require keys at least 512 to 1024 bits in length to be secure. MAC and symmetric encryption keys in general are considered strong if they are 128-bits in length. DSA and RSA are also orders of magnitudes slower that MAC and symmetric encryption algorithms. This slow down is largely because of the key sizes involved and the operations performed. The operations required to use public

key authentication are substantial and make it unsuitable for use with network attached storage.

Message Authentication Codes

Message Authentication Codes (MAC) provide both message authentication and integrity. A MAC is the result of applying a cryptographic one-way function to a secret key and and a message. The MAC is then appended to the message when the message is transmitted. Typically the same secret key is used to verify and generate the MAC. To verify, the receiver simply applies the same secret key as the sender and the message received to the function. If the result is the same as the MAC included with the message, the receiver is assured that the message arrived intact and was generated by someone in possession of the secret key.

Encryption functions such as DES can be used to implement one-way functions. However, plain one-way hash functions such as MD5 [49] or Secure Hash Algorithm (SHA) [58] can be used as a MAC function. The MAC function we us that is based on SHA and MD5 is called HMAC [30, 2].

MACs are useful since message integrity is important, and a MAC provides both message integrity and authentication in one calculation. HMAC, in particular, avoids encryption restrictions that are present in many countries.

One of the problems with MACs is that both parties involved in the communication must be in possession of the same secret key, so a method of key distribution must be employed. A key distribution scheme based on a trusted third party and symmetric encryption keys, such as Kerberos, may be employed. A public key distribution scheme can also be employed using public keys for encryption, not just authentication.

A novel scheme for authentication and key distribution based on HMAC called Kryp-

toKnight [5] could also be used. In form it is much like Kerberos. It uses a trusted third party and has tickets that are used to initiate communication between two parties. However, unlike Kerberos KryptoKnight uses only an elementary form of encryption (in the form of a one-time pad) that allowed IBM to get approval for its use around the world without restriction.

The session key is generated by the Trusted Third Party (TTP). Each party will receive the session key encrypted with a one-time pad that only the receiving party and the TTP can recreate. The one-time pad is generated by the TTP by generating a MAC for some of the communication data and a random string sent by the receiver. The MAC is not sent by the TTP, but is used as the one-time pad. Since both the TTP and the receiver share the secret used with the MAC, both can regenerate the one-time pad. This allows the reciever to decrypt the session key sent by the TTP.

KryptoKnight does not depend on synchronized clocks. Instead it uses one-time random numbers (nonces) to ensure the freshness of a message.

The simplicity and speed of HMAC algorithms, as well as their immunity from restrictions on encryption, make them perfect for use in network attach storage. The authentication scheme described in chapters 3 and 4 makes heavy use of HMACs and many of the concepts used in KryptoKnight.

CFS

The Cryptographic File System (CFS) [6] isn't actually a file system itself, instead it is a virtual file system that serves as an encryption layer between the user and a shadow file system. All the file data and directory data is stored encrypted on the shadow file system. The user provides keys to the CFS which it uses to decrypt files and directories from the shadow file system for reads, and encrypt files and directories for writes. There is a one-to-one mapping between the files on the

shadow file system and CFS. The shadow file system can be any local or network file system on the machine.

Keeping the data encrypted on the shadow file system allows for convenient management of backups. The backup administrators are able to copy the files to tape, but the data itself is not compromised. The confidentiality of the file system is also maintained if a distributed file system is used since CFS, in effect, provides end-to-end encryption.

There is some data that is not encrypted by CFS. File sizes, access times, and the structure of the directory hierarchy are all kept in the clear. Thus, CFS is vulnerable to traffic analysis attacks from real-time network data collection and file system snapshots. In addition, the authentication and integrity of the data is provided, in part, by the shadow file system. Old data can be replayed without detection, if the encryption key has not been changed. If the data itself does not have integrity checks the cipher text may be able to be changed without detection.

1.3 Overview of the Thesis

In the following chapters we present secure network attached storage that is able to leverage the benefits of extremely distributed file systems, such as xFS, without sacrificing security. The next chapter presents the object model, referred to as SCARED (Secure Array of Remotely Encrypted Devices), used by the network attached storage that will be leveraged throughout this thesis. The object model allows us to avoid the bottleneck of a management server for metadata, such as block location, access control, and cache coherency, as well as, the complexity of a distributed management protocol.

An authentication protocol for SCARED will be presented, followed by a cryptographic analysis of the protocol. SCARED protects the data from unauthorized access, without resorting

to intensive cryptographic operations. The protocol also allows the key distribution to be done independent of the storage device, making it possible to integrate secure devices into any existing security framework. The main advantages over capability protocols, such as NASD, are the ability to share access keys, identity keys, and user key derivation. Identity keys are especially advantageous because they reduce the number of keys managed by the clients and remove the need for a file server to create capabilities. The cryptographic operation are much faster than those used in public key cryptography. And the infrastructure requirements are much simplier than Kerberos.

Finally, a distributed file system, Brave, built using SCARED devices will be presented. Brave runs at each client, and provides file semantics for the data stored on the SCARED devices. The fundamental advantage of Brave over existing file servers is that it is serverless, so it is not limited by the scalability of a single server. It has this advantage without sacrificing security.

Chapter 2

The SCARED Object Model

In order to increase the aggregate bandwidth and processing to network storage, it has been proposed to directly attach storage devices to the network. This is in contrast to the more mainstream idea of putting the storage devices behind network file servers. Examination of past literature reveals that the idea of clients directly accessing storage servers is not new.

Previous file systems such as Zebra [22] and Swift [32] striped the file system across multiple network storage servers that the clients directly accessed. Zebra had a large block interface, called fragments, into which it logged client requests and Swift had a more object based abstraction. The Bullet [59] file system presented an extremely simple object based abstraction to interface to the network storage servers that stored immutable objects. More recently Petal [31] groups multiple storage servers into what is effectively a large block device onto which the clients map a distributed file system, Frangipani [54]. GPFS [25] and xFS [11], the serverless file system from Berkeley, use the clients in a cluster as storage servers to form a single distributed block device. NASD [17] and Trapeze [10], on the other hand, use an object abstraction with network attached storage. Of course this is only a small sample of current and past work, but it does show a long history of network

attached storage and their access methods.

This chapter addresses the advantages of an object interface over a block interface for network attached storage, and it proposes an object model that is used in our version of network attached storage. Given that most local storage interconnects, such as IDE and SCSI, are block based, it seems natural to have a block interface to network storage. We refute this intuition by presenting four areas in which an object interface has significant advantages over a block interface: additional semantics, storage allocation, caching, and authentication.

The next section describes some of the additional semantics that can be added to a network storage device to increase the overall efficiency of a distributed file system using network storage. Section 2.2 presents the allocation problems of network attached storage and their solutions using an object model. The advantages of object access with respect to caching and authentication are presented in §2.3 and §2.4. The chapter is summarized in §2.5.

2.1 Object Abstraction

As mentioned in the introduction, using an object model on the network storage allows additional semantics at the storage device. Some of the semantic information is simply a result of the object interface. For example, if a block interface is used, the relationship between disk blocks is not easily derived. However, in an object based model the network storage knows to which objects a block belongs and can optimize accordingly. One of the main optimizations of the Bullet [59] file system is to store all blocks of a file contiguously.

The object interface is even more advantageous than the block interface when functional semantics are added to the objects. The file systems reviewed in the introduction that had an object interface supported little more than the operations that can be done on normal file; namely

read, write, and truncate. These functional semantics are enough for storing files in objects, but if directory data is to be stored in objects, more operations need to be added to the object interface.

In the following sections, we present the operations for two object types: data objects, and meta-data objects. The data objects support operations usually found in an object interface to support file objects. The meta-data objects have additional semantics to support directories. The common attributes and operations of both types of objects will be presented in the next section before the operations specific to each object type are introduced.

2.1.1 Basic Semantics

Just as a block interface to storage uses block numbers, objects are identified by an object number, which will be referred to as the object identifier or OID. Clients use OIDs to access objects just as they would a block number to access blocks. An important difference is that the mapping of an OID to physical blocks requires additional meta-data that the mapping of block numbers to physical blocks generally do not need. This is because objects are not fixed size and the blocks they use will grow and shrink over time.

Another difference between block numbers and OIDs is that OIDs are created and deleted. In our object model, we do not reuse object OIDs. Since our OIDs are 128 bits we do not need to worry about exhausting our supply.

Our object abstraction also allows the storage to manage object meta data such as access times and size, as well as allowing clients to attach their own metadata to an object such as access lists. This allows the storage to more actively participate in a distributed file system, thereby removing much of the load from the distributed file servers.

When an object is created, the create request specifies which type of object to create.

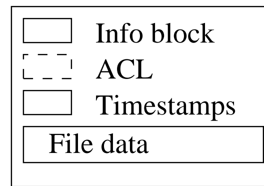

Figure 2.1: The structure of a file object.

In this chapter we outline two types of objects: data objects and metadata objects. We will first describe the data objects, which is the simpler of the two, followed by the metadata object.

2.1.2 Data Object Semantics

Data objects support the semantics associated with files in a file system. Basically a data object represents a logically contiguous set of data blocks. The storage device is responsible for mapping the set of blocks into actual physical blocks, which may or may not be contiguous.

Figure 2.1, shows a representation of the structure of a file object. Common to all objects are an informational block, the info block, that is accessed in its entirety by the clients and timestamps. If ACLs are used to restrict access to the object, an ACL will also be present. The data object also has file data associated with it. This is a variable length, logically contiguous set of blocks. They are accessed based on their offset into the data object. Blocks can be read and written. Writing past the end of object causes the object to grow. The objects can also be truncated. Using these operations and semantics, it is simple to map file operations onto data object operations.

Later sections will show the advantages of the object abstraction in terms of cache coherency, access control, and block allocations. In addition to these advantages, a storage device can also take into account the relationship between blocks that is inherent in the data object abstraction. Many studies [1, 43] have shown that the majority of file accesses are sequential. By knowing the

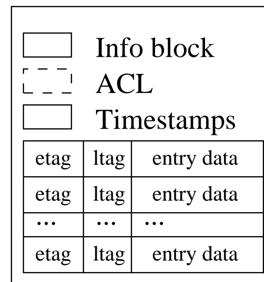

Figure 2.2: The structure of a metadata object.

sequential ordering of physical blocks in the object, the storage device can do read-ahead. Since accesses are done using an OID and not a physical block, the disk can reorganize physical blocks of an object to be contiguous, and move often used objects to the middle of the storage media in order to optimize performance.

Using the data object abstraction, the load on a file server can be greatly reduced by allowing clients to make data accesses directly to the storage devices. Many of the same advantages gained by moving file semantics to the network storage can also be seen by moving directory semantics to the storage devices.

2.1.3 Metadata Object Semantics

Since the semantics and operations of directories are so different from files, we have the metadata object type. A file system uses files to store that data of the file system. The files themselves are accessed through directories. The directories organize files into a hierarchal name space and provide location information about the files, or in other words, the data about the data – the metadata.

Figure 2.2, shows a representation of the structure of a metadata object. The structure is

similar to the data object except that instead of file data, a metadata object organizes the data by entries. This is because directory data is organized into directory entries in contrast to a contiguous set of blocks used to store file data. These entries are used to map names to objects, just as directory entries in a local file system map a name to disk blocks. When defining the abstraction for metadata objects, we wanted to ensure the data structures enabled the storage of metadata without dictating its structure. For this reason entries consist of a lookup tag, an entry tag, and variable length entry data. The tags are used to access the entry data, but entry data itself has no meaning to the storage.

There are two indices to the directory entries: the lookup tags and the entry tags. The lookup tag is used to optimize the lookup operation, which is one of the most common operations on metadata objects [60]. The lookup tag is set by the clients of the storage device and can be changed at any time. The entry tag uniquely identifies an entry in a metadata object and cannot be changed. It is used to identify the entry to be operated on by all the metadata operations with the exception of lookup.

When an entry is created in a metadata object, a unique entry tag is generated by the storage device to identify the new entry. This tag is not only unique at the time of creation, but will never be generated again for that directory object. As in the case of OIDs, there is no danger of running out of entry tags since the tags are 128-bit numbers.

The entry tag is also used to change the lookup tag and entry data of an entry and to delete an entry. When enumerating the entries of a metadata object, the entry tag is also used. Entry enumeration occurs when the client needs to list the contents of a directory. A storage device will try to put as many of the entries into a response to a request for enumeration as it can. If all of the entries cannot fit into a single response, the network storage will indicate to the client the entry tag of next entry in the enumeration.

By using metadata objects the load of the file server can be further reduced, if not eliminated entirely. The metadata abstraction provides enough semantics to allow efficient access to the entries, as well as allow access control lists to limit access to specific meta data operations.

2.2 Block Allocation

Using the object abstractions from the previous section we can simplify the management of block allocations. When storing data on network storage devices, care must be taken to allocate physical blocks on the disk in a consistent manner. The same blocks should not be allocated to different file system objects or file system data and metadata could get overwritten. By using an object abstraction at the storage device, the physical block allocations can be managed locally at the device.

When using block oriented network storage in a distributed file system, a distribute messaging protocol or a central server must be used to manage the allocation of blocks to the file system objects. For example, Frangipani and GPFS use a group messaging protocol, and Storage Tank [7] uses a server to manage allocation. Using group messaging protocols impose topology restrictions on the network. Not only must all clients be able to communicate with each other, but slow clients will affect the faster clients. The server based approach introduces network latencies to allocation requests, as well as another point of failure. All of these problems are solved by “centralizing” the allocation of blocks at the storage device.

As mentioned in the previous section, objects are logical entities as opposed to physical entities. Since the mapping of these logical entities to physical blocks is done by the device and not exposed by the storage device, block allocation is done transparently to the clients. The allocations can also be done atomically, since the block management is local. Finally the clients need only

communicate with the storage device to allocate storage, since all of the management is done locally at the device.

2.3 Cache Management

The cache management problem is very similar to the allocation management problem. Clients need to be notified when objects they have cached change so they can invalidate their cache. As with allocation management, cache management of block devices is usually done using a group message protocol or a cache management server. Conceivably, a block oriented network device could directly invalidate client caches, but a lot of information would need to be maintained per client to be able to know which blocks the clients have in their cache; this may impose a large memory requirement for storage devices. The object abstraction helps the cache management problem by providing a nice level of granularity of cache management.

It should be noted that not all distributed file servers invalidate client caches. Some, such as NFS, used a timer based approach to invalidate their cache. However, timer based approaches do not have good consistency guarantees or good performance characteristics.

There is a whole range of cache consistency models that can be implemented on top of the object abstraction. We chose a consistency model that has a small processing and memory overhead and still provides acceptable consistency and performance. Stronger guarantees, such as those of DFS, could also be implemented if sufficient memory were available.

Call-backs from the storage devices are used to invalidate client caches. Clients register interest in specific objects on the disks and are notified of changes. Whenever an object change has been committed to the non-volatile storage, a notification will be sent to interested clients. The specifics of the notification depends on the type of object. When a client removes an object from its

cache, it notifies the storage that it no longer is interested in the object.

Since data objects change often, and usually involve multiple updates, the notification for data object updates does not happen until changes have been committed on the storage device. This allows clients to send multiple write requests to the storage device before actually committing the changes to non-volatile storage. It also avoids having to send cache invalidation notices each time a write request is received. The invalidation notification will tell the client exactly which parts have changed so only those pages in the cache can be invalidated.

Changes to metadata objects happen much less often than changes to data objects in general [60]. So when a change happens, notifications are sent immediately to interested clients. These notifications will include the entry tags of the entries that have changed.

In addition to cache invalidation notifications, a version number is kept by the storage device for each object as well as each directory. The version number is incremented each time the object or an entry changes. This allows, for example, clients to revalidate their caches on a reboot. It also allows for conditional updates of objects and entries by allowing the client to request an update only if the version of the entry or object is the one the client expects. Since we do not have a locking mechanism at the network storage device, these conditional updates help to avoid consistency problems when simultaneous updates happen.

By using an object abstraction, we are able to provide a loose consistency protocol similar to AFS with good performance characteristics, while avoiding the overhead of group messaging protocols or another server. The low overhead of the protocol reduces the memory and computational requirements of the storage device.

2.4 Access Control

Just as the object abstraction allowed us to take into account the different object types to improve the cache management, we can use the object abstraction to control the kinds of access to the blocks on the storage device based on their object types. Access control is one area where the advantages of the object access over block access is very clear. Currently, block oriented access devices allow only course granularity of access control of the network storage. Usually clients are granted read or write access to whole partitions. This course grained access control is not sufficient when used in a file system. Access needs to be granted to blocks based on the object to which the blocks belong and the kinds of operations that can be performed on the object. The object abstraction allows us to do exactly that.

To control access to storage, the device must be able to either know what the client can do, or know on whose behalf the client is acting. Capabilities are used to convey to the network storage what the client can do. If the storage device is able to identify the client, an access list for the requested object is checked to grant access to the object. The network storage we have designed uses both of these access methods.

Capabilities are lists of access rights encoded in a block of bytes. The access rights cryptographically derived in such a way that the storage is able to validate their authenticity. The encoding allows the device to know the permitted operations and the targets of those operations. For example, a client may possess the capability that allows it read access to an object. When a client presents a storage device the capability, the device can validate the capability cryptographically and the check that the requested operation is permitted by the capability. Capabilities have the advantage that the storage does not have to know the identity of the client, so the decision to permit the operation is made quickly based solely on the capability.

One of the difficulties with using capabilities is distributing the capabilities to the clients. Because capabilities allow such fine grained access to objects on the device, there are a lot of them that can be generated and will need to be distributed. The other difficulty is revoking or "taking back" capabilities that a client possesses. Since a capability is just a block of bytes, the client can make as many copies as it wants, and the administrator revoking access cannot be sure the client has not kept a copy of the key.

Usually a capability needs to be revoked because the client lost access to an object. This kind of revocation can be avoided altogether by used access control lists (ACLs). When an object has an access control list, a client can be denied access by removing his identifier from the list. For this reason, it is often more convenient to use ACLs and identifiers instead of capabilities. Identifiers can also be advantageous to the clients since they reduce the number of keys a client needs to manage. The reason for the reduction is that instead of requiring a capability key for each object on a device that a client can access, it only needs a single key to identify itself to the device.

When identifiers are used, the storage device must maintain an access control list (ACL) for each object on the device. The ACL constitutes additional metadata that the device must track for each of its objects. The object abstraction readily supports ACLs since the granularity of access is at the object level. Just as each object on the disk is protected by an access control list, the disk itself is also protected by an access control list. The disk's access control list controls who can create objects on the disk and who can change the disk's access control list.

By using capabilities and identifiers with access control lists, we can provide access control for all the objects managed by the storage device. The ability to do this level of access control would be difficult if a block server were used, which explains why current distributed block servers do not do fine grained access control.

2.5 Summary

The SCARED object model is “as simple as possible, but not simpler”. It allows the clients simple abstractions to model both data and meta data file system objects. It also eliminates the need for client to client distributed messaging protocols or additional servers by centralizing the management of object allocation, access, and caching at the storage device.

The object allocations allow for efficient allocation of blocks at the disk without clients having to coordinate their activities. Future optimizations, such as block placement based on access patterns, can be done at the disk transparently to the client.

Not only does the object abstraction help with the allocation of blocks, it also enables them to be efficiently cached at the client. The caching policy we have presented allows clients to have cache consistency with very little overhead at the storage device. The small overhead property is a very important one since many of the network attached storage devices will have extremely limited resources when compared to conventional file servers.

Not only do objects help with caching and allocation, but they also help protect access to the data stored on the network. Since clients can access the network storage directly, the storage devices must be able to restrict access to their data. The next chapter will build on the object abstraction presented in this chapter to provide strong access protections to the data on the device.

Chapter 3

Deriving Keys for Authentication

From a security perspective, the big difference between a host attached storage device and a network attached storage device is that the former knows exactly from which host requests are coming. Requests to a network attached storage device can originate from any node on the network. In some cases, the network and hosts on the network are considered trusted, in which case the network provides information about the identity of the requester, but in general networks are considered untrusted. The most common networking protocol, TCP/IP, is vulnerable to a variety of attacks that illustrate the ease of faking the identity of nodes on an IP network [4].

The two most common ways of overcoming the identity problem are symmetric key based authentication schemes and public key based authentication schemes. Both of these schemes use a trusted third party to give out tickets or certificates to clients on the network to help identify themselves to other clients. Symmetric key based authentication schemes usually require a ticket for each pair of clients that are communicating; whereas, public key based schemes require only one certificate per client. The big disadvantage of public key cryptography is the computationally intensive operations that are involved. Both of these schemes are widely used in the form of Kerberos

[40] and Secure Socket Layer (SSL) [12].

While Kerberos and SSL could be used to fulfill the security needs of network attached storage, there are a few requirements that make it necessary to find a better approach to security. First, Kerberos has a large infrastructure associated with it. This implies that choosing Kerberos would force the network storage to only be deployed in a Kerberos environment. The large infrastructure also increases the administrative costs for each storage device. SSL also has an associated infrastructure, albeit simpler, that would also require the device to only be deployed in an SSL environment. In addition, the processing requirements make it unfit for low end network attached storage. Finally, both schemes require encryption in the device which means they are export controlled [55].

We have solved the problem by using one authentication scheme between clients and network storage, and another between the clients themselves. To overcome some of the problems mentioned above, we have devised an authentication scheme based on key derivation using one way hashes. These keys have identities and capabilities associated with them. The keys can be exchanged among the clients using whatever existing protocols are in place, *e.g.* SSL and Kerberos.

The key derivation, its associated protocol, and the object module explained in the previous chapter are collectively referred to as SCARED (Secure Array of Remotely Encrypted Devices). The next section explains the environment in which SCARED is used. Section 3.2 explains the method of key derivation. The way capabilities and identities are associated with the derived keys is explained in §3.3. Access revocation is discussed in §3.4. A security analysis of the derivation is done in §3.5, and §3.6 summarizes this chapter.

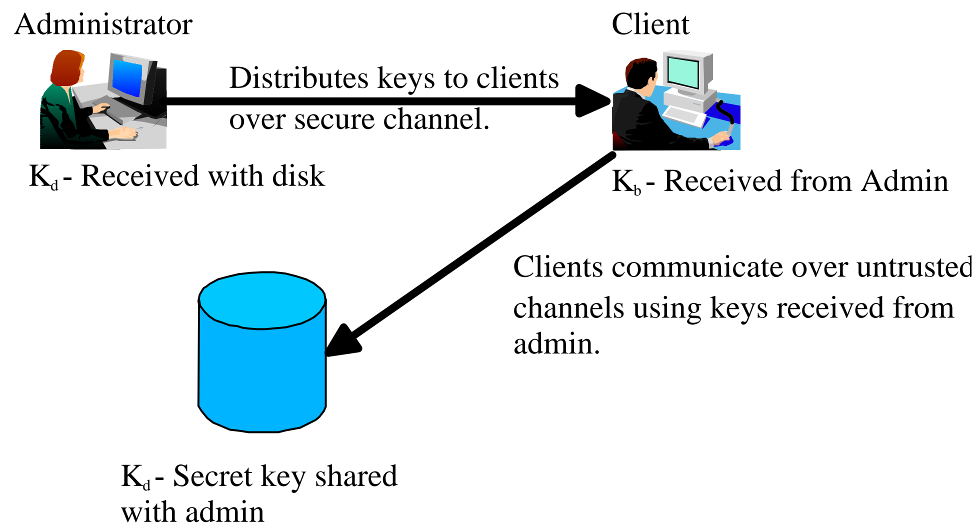

Figure 3.1: The administrator, the storage device, and the client are the three roles in SCARED. The key derivation scheme allows the administrator to generate access keys for the clients.

3.1 Distributed SCARED Environment

In the SCARED environment there are three roles: the client, the administrator, and the storage device. The administrator is the owner of the storage device. She controls access to the device. The clients use the storage device to store their data. SCARED's purpose is to enable the administrator to grant access to the network storage, and allow clients with access to share their access rights with other clients.

Initially, the administrator is the only one that can access a network storage device. When an administrator attaches the storage device to the network it will share a secret key with the device, which allows it to administer the device. The administrator uses this key to derive other keys for use by the clients. Clients use the derived keys to access the storage devices.

Figure 3.1 illustrates conceptually the three roles in the SCARED environment. Initially, the administrator will share a secret, K_d with the storage device. The administrator will use K_d to derive new keys. In this example, a new key K_b is derived and passed via a secure channel to a

client. The client can then use K_b to access the network storage over an untrusted network.

An important feature of the SCARED protocol is that the administrator does not need to be online with the disk when generating secrets for the clients. Not only does this relax the network topology requirements, but it also allows the administrator to give new secrets to the clients using off-line methods such as e-mail.

The SCARED communication protocol relies on shared keys to authenticate access to the devices. Not only must these keys be secret, but they must also carry information about the client in possession of the key, so that the storage device can check access. The next section explains the method of key derivation that SCARED uses, and §3.3 explains how the information used in key derivation is used to hold information about the key.

3.2 Key Distribution Without Key Exchange

We wanted to keep the device from having to do key management or be involved with distributing keys to clients, so the storage device itself knows only about one key: the disk key. This key is shared by the storage administrator and the storage device. It is the key upon which all other keys are based, and is used to bootstrap the security of the disk. We assume that the administrator receives the disk key with the storage device. This may be in the form of a smart card, disk, or paper that comes with the device. Another method, which is used by NASD, is to allow the administrator to generate and send the disk key to the disk when it is first connected to the network.

From this initial disk key we derive new secrets using a keyed one-way hash function, $H(D, K)$. The cryptographic properties of this function will be analyzed in §3.5, but for now three important properties should be noted. First, if K is secret, than the result of the function is also secret. Also, it is computationally difficult for an attacker to find K given $H(D, K)$ and D. Finally,

it is computationally difficult to find another D' and K' such that $K' = H(D', K)$ if K is not known. From an analytic point of view, we assume that $H(D, K)$ is a pseudo-random function [19] where D is the argument of the function and K is the key.

Using the keyed one-way hash function, an administrator can derive new keys for clients by hashing data, representing the attributes of the new key, using the disk key as the key to the hash function. If a client presents the data used to generate the key to the storage device, the device can regenerate the secret since it is in possession of the disk key. Clients can also generate new secrets by hashing new key data using a key in their possession. These new keys can then be regenerated by the disk given all of the data associated with the keys from which they were derived.

In order for keys to be meaningful to the storage device, they need to have some data associated with them to convey identity and capability along with other data associated with the key. The hash function binds the data associated with a key, referred to as the public key data, to the key itself.

The public key data allows the storage device to derive not only the key the client is using, but also to check the access the client has to the device. Because the key is derived using a one-way hash and the key data, when a key is used by a client, the client must also send the key data associated with the key. The binding between the key and key data allows the administrator to put information in the key data that the storage device uses to grant access to the client. By including a expiration date as part of the key data, the administrator is also able to limit the lifetime of the key.

3.3 Key Types

The authentication needs of a client and storage device differ, so the keys they use also differ. The client needs to verify the responses received from a storage device actually came from a

given device. The device needs to verify that the client has the authority to make a request. When a key is used by a client to send a request to the storage device, we refer to the key as an access key. A key used to verify the origin of a response, is referred to as a response key.

Another way of classifying keys is by the type of public data associated with them. If the data associated with a key has to do with the type of operations that can be done using the key and the targets of the operations, the key is referred to as a capability key. If the data has to do with the identity of the possessor or group membership, the key is referred to as an identity key.

Both capability and identity keys can be used as access keys. If the objects have access lists associated with them, the device will use identity keys to check access. If access lists are not used, the device must check access using capability keys. Access lists imply fewer keys to be managed at the clients, but more meta-data to be managed at the devices. Capability keys require very little meta-data to be managed at the devices, but more keys to be managed by the clients.

Since the clients are only interested in authenticating the device that generated a response, response keys are always identity keys. A client receives a response key generated specifically for that client by the administrator to authenticate responses from a specific device.

3.3.1 Generating Capability Keys

A capability key allows a specific operation to be performed on a storage device. The type of operation permitted and the details of that operation are governed by the data used to generate the key.

The key given to the client is generated by hashing the disk key with the key data. The result of the hash is the capability key. The capability key and the data corresponding to the capability key are given to the client. Note that the capability key must be kept secret so a secure channel must

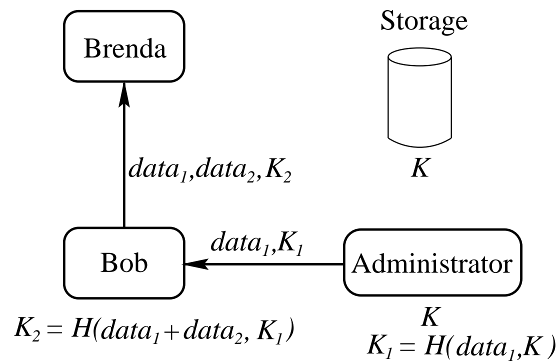

Figure 3.2: The administrator shares a key, K, with the storage device which is used to generate keys to be given to the clients. In this example the messages must be exchanged over secure channels.

be used to send the key to the client.

A capability key may be used to generate another capability key that is a restricted subset of the capabilities of the first key. This can be done by anyone in possession of a capability key, not just the administrator, which makes it convenient for highly distributed file systems. When distributing the new capability key, the new key-data corresponding to the new key includes the data used to compute the new key and the key-data from the original capability key.

For example, in Figure 3.2, if the administrator wishes to grant Bob the ability to read and write object 232 on the storage device, the administrator would generate K_1 with the READ and WRITE attributes in $data_1$ along with object 232. Bob could then grant Brenda the ability to read object 232 by only including the READ attribute and object 232 in $data_2$. Brenda could generate another capability key to read object 232, but could not generate a capability key to write to object 232, since the WRITE attribute is not among the capabilities of the key that Brenda possesses.

3.3.2 Generating Identity Keys

Identity keys allow a receiver to check the identity of the sender by including an identification string as part of the key data. As was done with the capability keys, identity keys are

generated by hashing the identification string as part of the key data and the disk key. The resulting identity key, and the corresponding key data, are given via a secure channel to the client.

As with capability keys, identity keys can be used to generate other identity keys. When a new identity key is generated from another, the entity in possession of the original key is vouching for the identity of the entity for whom the key is generated. This allows a non-administrative user to create a new identity key to allow access to objects the user can already access.

For example, in Figure 3.2 if the administrator wishes to identify Bob to the storage device, the administrator would include a string identifying Bob in $data_1$. Bob could then create a new key identifying Brenda to the disk by including a string identifying Brenda in $data_2$. It should be pointed out that the storage device would only recognize K_2 as valid if Bob were authorized to identify other users, or Brenda is only accessing objects that Bob can access.

When identity keys are used, the storage device must maintain additional meta data at the objects to be able to check the operations that a given identity is allowed to perform. This extra meta data is not needed when using capability keys, since the key data specifies the operations the client is allowed to perform.

3.3.3 Combining Keys

Figure 3.3 illustrates an interesting use of deriving a capability key from an identity key. In this example, the user is in possession of an identity key and would like to print a file. The user can generate a capability key, K'_{bob} and send it to the printer. Because of the capabilities used to derive K'_{bob} the printer can only access the object 123 if Bob has access to that object. The rest of the objects to which Bob has access remain inaccessible to the printer.

When the disk receives the read request, it will see that K'_{bob} is being used and will receive

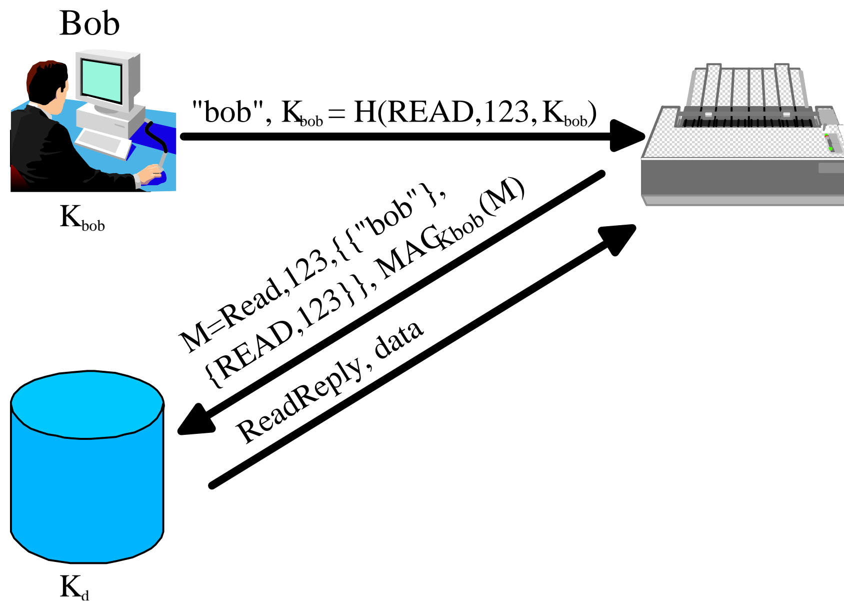

Figure 3.3: Mixing identity and capability keys to enable printer access to a data object.

the key data corresponding to that key. Because the first part of the key data consists of an identity, it will check the ACL of 123 to insure that Bob can read the object. If he does have access, the device will then check that the capabilities present in the second part of the key allow the read operation on 123.

Mixing capability and identities prove to be very useful when allowing proxy operations with another device that does not have an identity associated with it. Other examples are backup and archive services, third party data mining and processing, and third party transfers. A key enabler of these applications is the ability for non-administrators to derive capability keys using keys in their possession.

3.4 Revocation

With all these keys being generated, it is important to be able to disable or revoke a key if it is compromised. Obviously, the best way to deal with the problem of key revocation is to make the keys secure. Smart cards and tamper resistant chips are some of the ways of making the keys “secure”. However, the smart cards themselves can be lost, which would again necessitate the revocation of the keys in the cards.

SCARED implements three ways of revoking keys. First, keys have a limited life time. Second, valid keys are controlled at the target. Third, all keys can be revoked for the storage device by changing the disk key.

Only access key revocation needs to be done at the storage device since response keys do not need to be revoked. Response keys are used by the client to authenticate responses from the disk, so the client simply stops using a key that has been revoked. The response key does not have any access rights associated with it, so an attacker would not be able to gain access to a storage device using a revoked response key. No client would recognize responses using the revoked key, so an attack against a client with a revoked key would also be useless.

3.4.1 Key Expiration

When an administrator gives a key to a client, the administrator can include an expiration time in the key data of the key. Given that a key can only be used at one target, the expiration time is relative to the timer on that target. By using relative time, the need for synchronized clocks is removed. The expiration time will limit the lifetime of the key.

If an attacker is able to compromise a key, the key would only be useful until the expiration time. If the expiration time is kept short, the attacker will only have a small window of opportunity

to exploit the key. The ability to expire a key is also useful when using revocation lists because they keep the list from growing without bounds. When a key is expired, it can be removed from the list.

3.4.2 Capability Key Revocation

To aide in capability key revocation, we associate salt to a capability key. *Salt* is a number, much like a nonce, that will never be changed to a value it has had previously. It is not considered secret and it is stored with every object or meta-data entry. When a capability key is generated for an object or entry, the salt of the object or entry must be included in the key data. When the key is used, the salt in the key data must match the salt in the object or entry being operated on.

Capability keys for an object or entry can be revoked by changing the salt at the object or entry. When the salt is changed, all of the keys that included the salt will be invalidated, since the salt in the keys will be different from the new salt.

3.4.3 Identity Key Revocation

Identity key revocation can actually be done in two ways. The first uses revocation lists for unexpired and invalid identities. The second method is a simpler revocation scheme that requires the storage device to know *a priori* the identity of clients with which it will be communicating.

When key expiration information is present in the key data, only keys that haven't expired need to be revoked. If it is assumed most keys that are not expired are valid, then an efficient way of revoking keys is to give a list of key revocations to the storage device. Based on the previous assumption, the revocation list should be short so the identities present in requests to the disk could be checked against the list before accepting them as valid. Once a revoked key is expired, it would be removed from the revocation list to keep it from growing without bound. In theory, revocation

lists could be used with capability keys. However, given that the number of capability keys will be on the order of the number of objects on a SCARED device, the list could grow extremely large.

The second way of doing identity based authentication is to include a counter in the identity key calculation. The counter is then stored in a table on the storage device indexed by the client id. When a client makes a request, the device verifies that the counter in the table is less than or equal to the counter included in the key data of the request. If the counter in the table is less than the counter in the key data, the counter in the table is set equal to the key data counter. To revoke a key, a new key needs to be generated with a new counter. When the new key is used, the table will be updated and the old keys will become invalid.

3.5 Security Analysis

The previous sections have presented a way of deriving new secrets based on an initial master secret shared by the storage device and the administrator. The new secrets are shared by the storage device by exchanging only public information about the secret and not the secret itself. In addition, the method of derivation allows data to be bound to the new derived secret. In this section we seek to prove that only authorized parties can derive new secrets and that the data that is bound to the new secrets cannot be changed in a way that is undetectable by the storage device.

To begin our analysis, we must first more formally define the keyed one-way hash function introduced in §3.2. The cryptographic concepts used in this chapter and the next are more formally analyzed in [20, 33].

Pseudo-random functions are the basis of our key derivation. Informally, a pseudo-random function cannot be distinguished from a random function by a party, the adversary, that does not possess the secret used to compute the function. As the secret used in the function de-

creases in size, it is more likely that the adversary is able distinguish the function from a random function.

To define pseudo-random functions more formally, we must talk in terms of probabilities and probabilistic polynomial time machines. Appendix B formally defines pseudo-random functions. An integral part of the pseudo-random function is the key, which we refer to as the secret, that determines the output of a pseudo-random function for a given input. The size of the secret, the *security parameter*, of the function determines the probability of successfully appearing random to the observer. The following fact follows directly from the definition.

Fact 1. *A pseudo-random function, prf, has the property that*

$$\forall G \, \forall c > 0 \, \exists n_c \, \forall n > n_c \, \forall x \, Prob[G(x) = prf_S(x)] < n^{-c}$$

where the probability space is over choice of S and internal coin flips of G, and where G is a probabilistic polynomial time machine, and $n = |S|$ is the security parameter.

Using this fact, we can now define the derivation function that we used to derive new shared secrets.

Definition 1. *We define $H_K(P) = prf_K(P)$ where K is a secret and P is public and prf is a pseudo-random function. A pseudo-random function cannot be distinguished from a random function by an adversary not in possession of K in polynomial time with non-negligible probability.*

We do not restrict the adversary, specifically, the adversary can see past $\langle P, H_K(P) \rangle$ pairs and may obtain other $\langle P, H_K(P) \rangle$ pairs from other clients or administrators in possession of K.

As a precondition both the disk administrator and the storage device share a secret K_d. The disk administrator must be able to generate new secrets that are shared only by the administrator and the storage device without using a key exchange protocol. We will first show that the key

derivation method allows the administrator to create new secrets for clients and bind capabilities to those secrets. Then we will show that the clients themselves can create new secrets with capabilities bound to them. There is one claim which follows directly from the definition of $H_K(P)$ which we will now state.

Claim 1. *$K_a = H_K(P_a)$ for a given P_a cannot be computed in polynomial time with non-negligible probability by an adversary who does not possess K. Further, the adversary would not be able to distinguish between a random K_a and $K_a = H_K(P_a)$.*

Proof: While this claim follows directly from the definition of the pseudo-random function, it is interesting to note the following contradiction. Let us assume that given P_a the adversary can compute K_a without K with non-negligible probability. And adversary would be able to distinguish H from a random function with non-negligible probability by generating a K_a for a P_a and checking if H outputs K_a. Since H is pseudo-random the adversary cannot generate K_a or even distinguish from a random K_a. □

It is not enough that the adversary cannot generate K_a since it might be able to derive a few bits or a relation on some of the bits. For this reason we also needed to claim also that the adversary could not even distinguish the new key from a random key. This means that even individual bits or relations amoung bits cannot be discovered by the adversary.

There are two ways to view the relationship between K_a and P_a. First, because of the way K_a is derived, K_a authenticates P_a to someone in possession of K. This is how the construction is used in MACs. We have chosen to view the relationship as P_a describing K_a. As will be shown in the following theorems, if K_a is used as a secret to access a storage device, the derivation of K_a allows the administrator to describe K_a using P_a. We capture describe this relation by saying that P_a is associated with K_a.

Definition 2. *We say P_a is associated with K_a if K_a cannot be used without P_a and P_a describes the capabilities of K_a.*

The techniques used in §3.3 depend on P_a being associated with K_a. It is because of this association that we can encode attributes describing K_a in P_a. The following theorem describes the association.

Theorem 1. *If a client presents $K_a = H_{K_d}(P_a)$ to a SCARED device that has K_d, the device will be able to reproduce K_a and verify with overwhelming probability that the administrator associated P_a with K_a.*

Proof: Because H is a well know function, P_a is public, and the device is in possession of K_d, the device can calculate K_a by applying H to P_a and K_d. Claim 1 says that only someone in possession of K_d could compute K_a for P_a. Since only the administrator and disk share K_d, K_a must have been generated by the administrator using K_d. (Note, we are assuming that clients and administrators don't let their secrets be compromised.) The administrator encodes the capabilities of K_a in P_a. Since only the administrator could have generated the pair $\langle K_a, P_a \rangle$, P_a must be associated with K_a. $\square$

This theorem allows the disk administrator to create capability keys or identity keys by including them in the public data. Other attributes including expiration times can be included in the public data. Since the administrator creates the public data, it can be used to convey information about the key to the device.

Using theorem 1 we have proven that the disk administrator can derive keys for clients and bind capabilities to those keys. To allow a greater degree of delegation of access we need to prove that clients can derive keys for other clients using keys they possess.

Theorem 2. *If a client is in possession of K_a, which is a secret that can be derived by the SCARED device, and P_a, which is the public data associated with K_a, the client can generate a new secret $K_b = H_{K_a}(P_b)$, where P_b is some public data, such that the SCARED device can reproduce K_b and verify with overwhelming probability that P_b and P_a are associated with K_b.*

Proof: Because the device can derive K_a from P_a, the device can apply P_b and K_a to H to derive K_b. Just as in theorem 1 since H is a pseudo-random function, K_b must have been produced using P_b and K_a, thus P_b is associated with K_b by a client in possession of K_a. □

Using this theorem any client in possession of a key can derive keys to be used by other clients. By binding new public data created by the first client to the key, the first client can restrict what the second client has access to. Since the first client's key also has public data associated with it, the device can verify that the second client cannot use the key for something that the first client did not allow. The public data of the first client's key is also bound to the key generated for the second client, so the device can verify that the first client did not delegate more access than it had to delegate.

Using Theorem 2 we can now generalize Theorem 1 to apply to derived keys.

Theorem 3. *If a client presents $K_b = H_{K_a}(P_b)$ to a SCARED device that has K_d, the device will be able to reproduce K_b and verify with overwhelming probability that P_b was associated with K_b by a party whose access is described by P_a.*

Proof: We prove this theorem by induction on the number of derivations from the initial key received from the administrator. The base case is a key K_a has been derived from K_d by the administrator.

Induction Base: K_a received from administrator.

By Theorem 2, the SCARED device can reproduce K_b and verify that P_b and P_a is associated with K_b. By Claim 1, K_b can only be produced by the party in possession of K_a. Theorem 1 and Definition 2 says that P_a describes the access of the party that possesses K_a. Therefore, P_b was associated with K_b by a party whose access is described by P_a.

Induction Step: Assume true for derivations of depth less than or equal to $n-1$. K_b has a depth of n.

The proof for the induction step is the same as the induction base except that instead of using Theorem 1 we use the induction hypothesis. Since K_a has a derivation of depth $n-1$, the induction hypothesis shows that P_a is associated with K_a. □

Theorem 1 shows that the administrator can create a secret for a client that it shares with the storage device and at the same time associate data with that secret for use with the storage device. With Theorem 3 we have generalized Theorem 1 to include keys derived by clients. These two theorems allow us to encode access information about the clients in the keys they use, so that the device may grant access based on this information. We will use these theorems more in the next chapter.

3.6 Summary

As network storage becomes more pervasive the importance of authentication will become even more evident. The current public key and symmetric key methods of providing authentication information to network servers requires too much overhead and infrastructure for use with network attached storage. The key derivation scheme presented in this chapter offers a way of providing strong authentication information to network attached storage without a lot of infrastructure or computational intensive operations.

The SCARED protocol uses the key derivation scheme to convey information about the clients, as well as set up shared secrets for use by the SCARED wire protocol. The next chapter presents the SCARED wire protocol that will build on the concepts introduced in this chapter. Together the derivation scheme and the wire protocol will be used as the foundation for an authenticated serverless distributed file system.

Chapter 4

An Authenticated Message protocol for SCARED

When files are accessed by network clients, the requests must be authenticated and permissions checked before the access is allowed. The file servers generally do this kind of checking. However, if clients are allowed to directly access the disks, the disks must also be able to verify the authority of the client's access to the data.

The authentication protocols presented in the next sections use the object abstraction and key derivation scheme of the previous chapters to implement authentication protocols that do not require encryption and synchronized clocks, while allowing for delegation of authority and shared keys that are necessary for building a serverless file system.

The SeCure Authentication for Remotely Encrypted Devices (SCARED) protocols were developed at IBM research for use in network attached storage. One of the main design requirements was minimizing the management overhead of the storage devices. File servers require a substantial investment in management resources. By pulling the storage out of the servers and

network attaching them, the number of managed network devices increases. If the administrative requirement increases proportionally to the number of devices, the system would quickly become unmanageable. The management of network attached storage is further complicated due to the lack of a management console with a keyboard and display. For these reasons we push the administrative overhead out to the clients, where the administration of the storage device can be done along with the normal configuration of the client to use the network storage.

Storage devices are deployed in environments with a wide variety of existing authentication systems, such as Kerberos and public key based systems, so we did not want to assume too much about the environment in which the devices are deployed. The authentication operations done at the storage device are simple, and allow the device to be oblivious to the security environment in which it exists. Since keys used to interact with the storage devices are generated and exchanged by users and administrators without having to communicate with the storage, the key exchanges can take place within the existing systems.

SCARED addresses authentication. We believe the confidentiality requirements of storage devices is best solved by encrypting and decrypting at the clients. Encrypting data is expensive in terms of processing overhead and introduces latency. By doing the encryption and decryption at the client, the data is encrypted over the network and on the storage media itself, without any overhead at the server. SCARED does not preclude link level encryption. Section 4.4 presents how encryption keys can be negotiated for use in link level encryption.

Chapter 3 introduced the three roles in SCARED: the client, the administrator, and the storage device. The storage device shares a key with the administrator. The administrator uses this key to generate other keys for use by the clients. Clients use the derived keys to access the storage devices.

An important feature of the SCARED protocol is that the administrator does not need to be online with the disk when generating secrets for the clients. Not only does this relax the network topology requirements, but it also allows the administrator to give new secrets to the clients using off-line methods such as electronic mail.

SCARED addresses three aspects of security: identity/capability, integrity, and freshness. When a message is received, the recipient needs to validate who the message was sent by or at least that the sender was authorized to send the message. Next, the receiver needs to validate the integrity of the message, or in other words, that the message was not changed in transit. It would seem that being able to validate the sender would imply that the recipient is also able to validate the message was the message sent by that sender, but in practice this integrity guarantee is not always available. SCARED enables the network storage to validate both the identity of the sender and the message that was sent by the sender. Finally, the receiver must be able to validate that the message was sent recently (or that the message is fresh), or at least validate that the message is not a replay of an older message.

The next section will present the method used by SCARED to provide identity and integrity guarantees. Section 4.2 shows how freshness guarantees are done. The request and response protocols are described in §4.3. A security analysis of the protocol is presented in §4.5. This chapter is summarized in section 4.6.

4.1 Integrity and Identity Guarantees

Since we assume clients and storage devices communicate over untrusted channels, both parties must be able to verify the identity of the originator of a message and that the message was not changed in transit. Both of these requirements are satisfied by using a cryptographic construct

called a Message Authentication Code (MAC). The specific construction we use is based on a one-way hash and is referred to as HMAC [30]. [2] cryptographically analyzes the strength of the HMAC construction.

A MAC function takes a string and a secret key and outputs a fixed length string. The MAC has some cryptographic properties that allow either party to verify that the message sender was in possession of a specific key and that the message was not changed in transit.

A precondition to using a MAC is that both parties are in possession of the same key. If we assume two parties, A and B, wish to exchange a message, M, using a key, K, a MAC, C can be computed by both parties using $C = MAC_K(M)$. The MAC is used by attaching the computed MAC to the message being sent. For example if A sends M to B, A would send the following:

$$A \rightarrow B : M, C$$

Note that K is not sent over the network and the MAC function preserves the secrecy of K when used to calculate C. When B receives M, B can recompute C since it is in possession of K. If C equals the recomputed MAC, $MAC_K(M)$, B will know that M was sent by someone in possession of K.

Using a MAC with keys derived according to the SCARED protocol requires that more information is transmitted since the storage device does not directly possess the key used by the client. For example, if a client, A, is in possession of a key, K_A, and the data associated with it, P_A, A could send a message to the storage, S using the following protocol:

$$A \rightarrow S : M, P_A, C$$

The storage device can derive K_A since $K_A = H(P_A, K_D)$, where K_D is the disk key, since the device is in possession of K_D and P_A was sent by the client. Once the device has derived K_A,

C can be recomputed to check the MAC. Because P_A is bound to K_A, the storage device knows information about the client in possession of K_A as explained in chapter 3, so the storage device can check the capability of the client to take the requested action.

4.2 Freshness Guarantees

Integrity and identity guarantees are not sufficient for an authentication protocol, since older messages can be replayed without detection. Replayed messages will have valid MACs since they were sent by the client and have not been modified. Freshness guarantees allow detection of message replays by ensuring the messages have been sent in the recent past, or were in response to a pending request.

The first phase of the SCARED protocol is to establish a freshness guarantee. After a freshness guarantee has been established, the clients and storage use the message protocol to send responses and receive replies. When presenting the protocols, it is assumed that the clients are in possession of the keys needed for accessing the storage, and that the storage is only in possession of the disk key. The access key used by the client is denoted by K_a and the key data corresponding to K_a is denoted by P_a. The response key is denoted by K_r and its key data P_r.

To guarantee the freshness of messages, SCARED uses timers, nonces, and counters. When using timers, all parties involved in a transaction have timers that are reasonably synchronized. Nonces and counters do not require any kind of clocks, only that the nonce and counters never take on the same value. Counters are also required to be monotonically increasing.

The clients always use nonces to check the freshness of a response since a nonce is a freshness guarantee with the fewest requirements. When illustrating the protocol exchange, the client nonce will be denoted using F_c.

Storage devices require clients to include a timer or counter in the request to check the freshness of the request. Since the client must be able to calculate the freshness guarantee that the device is using, nonces cannot be used. If the communication with the device is session oriented, the device can key a counter synchronized with the device based on the number of messages sent. Otherwise, a timer must be used.

The storage counter or timer will be denoted using F_s. In the following message exchanges $FGRequest$ and $FGResponse$ correspond to constants used in the communication protocol to indicate the request and response of a freshness guarantee. Before making requests to the storage, the client must request the storage counter or nonce using the following protocol:

$$C \rightarrow S: \quad M = \{FGRequest, F_c, P_r\}, \mathrm{MAC}_{K_r}(M)$$

$$S \rightarrow C: \quad M = \{FGResponse, F_c, F_s\}, \mathrm{MAC}_{K_r}(M)$$

When the storage receives the request in the first message, the storage is able to generate K_r using P_r as shown in §3.2. If $\mathrm{MAC}_{K_r}(M)$ as calculated by the storage device matches $\mathrm{MAC}_{K_r}(M)$ included in the request, the device knows that M was generated by a client in possession of K_r, so it will generate a response using K_r. F_c is copied unchanged by the storage device into the response.

When the client receives the second message, it is able to check the MAC since it is in possession of K_r, and thus know that it came from the storage. The presence of F_c in the response allows the client to know that the message is in response to the first message. After the message exchange the client is in possession of F_s, the server freshness guarantee, which it uses to establish the freshness of future communications with the server.

4.2.1 Verifying Freshness using Counters

If the communication with the storage is session oriented, counters are convenient to use for checking the freshness of requests since they do not require clocks. At the beginning of the session the client will obtain F_s, the initial session counter. Each time the client transmits a packet, it includes the counter in the request and increments the counter for the next request.

The device is able to verify the freshness of the request by ensuring that the request includes a counter that is one greater than the previous request from the client. This implies that the storage device must be able to maintain a counter for each active session. The initial counter sent to the client must be generated in such a way that a counter used in a session with the device by the client was never used in a request by any client of the device in other sessions. In our implementation the counter is 128-bits, so the SCARED device generates a 64-bit nonce, based on the current time, for the high 64-bits of the counter, and initializes the lower 64-bits to zero.

4.2.2 Verifying Freshness using Timers

If the communication with the storage device is not session oriented, timers are used to allow the device to check freshness without having to keep freshness information about all the clients. To use timers, all clients intending to communicate with a storage device need to synchronize their timers with the timer of the storage device. This is done in the first phase of communication with the disk by setting F_s to the current device timer.

The client synchronizes its timer with the storage device by saving the difference between its timer and the storage device's timer. Since the client maintains a delta between its timer and the device's, the storage devices need not, and in most cases will not, have synchronized timers. The client includes the device's current timer in all requests to the storage device. This enables the

devices to check that the message is fresh in the sense that it was sent recently.

Because of network latencies, clock drift, and the latency of responses to requests, checking timestamps alone does not provide a strict guarantee of freshness. In particular an attacker could replay transactions in a small time window. To thwart the recent-past replay attack, a list of message authentication codes used in the recent-past are kept and checked with each message. If the code exists in the list, the message is considered a replay.

To compensate for clock drift, the storage device includes its current timer in all responses. The clients can then resynchronize their timers each time a response is received from the storage device.

4.3 The Request/Response Protocol

Once the client is in possession of the server freshness guarantee, generic requests can be made to the server. This section presents the generic request and response protocols used by the clients to communicate with the network attached storage devices.

4.3.1 The Request Protocol

The client request has the form:

$$C \rightarrow S : M = \{Operation, data, P_a, P_r, F_c, F_s\},$$

$$\text{MAC}_{K_a \oplus K_r}(M)$$

The operation requested and the data that goes with the operation are followed by the key data for the access and response keys that are used in this communication with the network storage. The device is able to regenerate the K_a and K_r using P_a and P_r, so that it can verify the MAC. F_s

is included to ensure that the message is fresh, using either the counter or timer based techniques explained in the previous section.

If the MAC is valid, the device knows that the message arrived intact and that it was sent by a client in possession of K_a, but it still must verify that the client is able to request the operation. The two approaches used by SCARED to check access are identity based and capability based. In identity based systems, the disk needs to be able to check access based on the identity of the requester. In capability based systems, the disk is only interested in the ability of a requester to perform a transaction.

Capabilities are granted by the administrator or a client in possession of a capability by generating an access key, K_a, with the capabilities contained in the key data, P_a as explained in §3.3.1. Therefore, the client in possession of K_a is also in possession of the capabilities listed in P_a. Since K_a may be derived from other access keys, the disk must ensure that each time the key is derived from another key, the capabilities in the key data of the derived key are a subset of the capabilities of the original key. To check if the client is able to carry out the requested operation, the device checks that the operation requested is listed as one of the capabilities.

If identities are used, P_a will contain the identity of the requester. In order for the disk to check the ability of a requester to perform an operation, the disk must maintain access lists on each object. When a request arrives, the identity in P_a is checked against the access list of the requested object to see if the client can request the operation.

4.3.2 Response Protocol

The authentication needs of the client are quite a bit simpler than the needs of the disk, since it only needs to verify the response was sent by the disk in reply to the client's request. The

device response has the form:

$$S \rightarrow C : M = \{Response, data, F_c, F_s\}, \mathrm{MAC}_{K_r}(M)$$

K_r is used in the MAC since it is the secret shared by the client and disk. The capability and identity keys may be shared by different clients, but the response key, K_r will only be held by one of the clients. After validating the MAC, the client will know that the response arrived intact from the disk. The presence of F_c allows the client to check that the response is for the request that also included F_c. F_s is included to compensate for clock drifts if timers are used by the disk.

The key data are not included in the response since the requester must already be in possession of K_r.

4.3.3 Asynchronous Responses

With one exception, all messages sent by a SCARED device are in response to a request that originated at the client. The exception is the cache update callback. This asynchronous message is sent when another client has committed changes to an object cached on the client. The message arrives in the form of a response, as described above. Since there is no request that corresponds to the callback, we are left with the problem of determining a client freshness guarantee to put in the response.

Since the guarantee must be based on something chosen by the client, the only thing we can use is a freshness guarantee of a previous request. As it turns out, we can simply reuse the freshness guarantee of the last response to the client. It is easy for the client to remember the last guarantee that it received in a response. To avoid replays, the client must keep a history of MACs used with the last guarantee. As long as each MAC is different, the client can be sure that the asynchronous response is not a replay.

4.4 Encryption

A key feature of a secure distributed file system is the confidentiality of file data. Currently, of the commercial distributed file systems, only DFS [15] has the option of encrypting data exchange between client and server.

A stronger level of data privacy can be obtained if the data is encrypted by the client and sent to the server to be stored in its encrypted form. This kind of client side encryption is done by the Cryptographic File System [6] (CFS), which encrypts data before being stored in a shadow file system and decrypts the data as it is read. Using CFS with SCARED would keep the data confidential and avoid the performance impact of encrypting at the storage devices.

CFS has a key distribution problem, since the encryption keys must be remembered and distributed by users. To overcome this problem, we propose storing the encryption keys in the meta-data encrypted with group and user encryption keys. This allows keys to be obtained at the moment they are needed.

One of the problems with storing the encryption keys in the meta-data is that if group or user encryption keys are changed, all the meta data needs to be updated by re-encrypting the keys using the new group or user keys.

If the storage devices are trusted to keep data confidential, the problems with encryption key distribution can be avoided by encrypting and decrypting at the storage devices. To encrypt the data between the clients and storage devices, they must share an encryption key. They already share a response key, so an encryption key can be generated by rehashing the response key with a public constant; but requiring the storage device to do link level encryption increases the processing requirements of the device.

Whether or not the network storage is involved in ensuring the confidentiality of the data,

the SCARED protocol satisfies the authentication requirements of network storage.

4.5 Analysis of Message Protocol

The previous analysis shows that a derived key received by a client is a secret shared with the storage device. In addition, the analysis also shows that the key data associated with a key is bound to the key in such a way that when a client uses the key, the device can verify the attributes associated with the key.

In this section we will analyze the two message exchanges used by SCARED: the freshness guarantee exchange, and the generic message exchange.

There are two keys used in the protocols: the access key, K_a, and the response key, K_r. The access key is used to make requests and has access rights of some form bound to it. The access key may be shared by other clients who may not necessarily trust each other. The response key is used to verify the origin of a response and is shared with, and received from, trusted administrators and clients.

To begin our analysis we must formally define MACs. In our definition we define the properties of the MAC function that we use. Other properties are defined in [46].

Definition 3. *We define $MAC_K(x)$ as a pseudo-random function with the specific property that given a message, M, an adversary without K would have a low probability of finding in polynomial time a C such that $C = MAC_K(M)$. This property holds for an adversary that is able to see past messages, not equal to M, and the resulting MAC.*

To ease the wording of the proofs, we will use the term *computationally feasible* to refer to an operation that can be computed in polynomial time and with more than negligible probability

of success. We also use the phrase *with overwhelming probability* to refer to a probability over the choice of keys used in the MAC function that is negligibly less than 1 for every probabilistic polynomial time algorithm.

In the following proofs the adversary is allowed to watch, modify, and insert into the messages between the client and the storage device. The adversary is also allowed to be another valid client or storage device. It should be noted that an adversary that is another storage device would possess and different K_d than target real storage device. An adversary that is another client may possess a K_a that is shared by the real client if it shares a capability or an identity with the real client (such as belonging to the same group), but an administrator gives a unique K_r to each client, so the adversary and the real client cannot share K_r. The administrator is trusted an cannot be an adversary.

4.5.1 Exchanging the Freshness Guarantee

The first exchange between a client and server must be a request for the server's freshness guarantee, denoted F_s. The client includes a nonce, F_c generated for the request. The request is MACed using a key, K_r, and includes the public data, P_r, associated with K_r. The server responds with F_s and includes F_c MACed with K_r as follows:

$$C \rightarrow S: \quad M = \{FGRequest, F_c\}, P_r, \mathrm{MAC}_{K_r}(M) \tag{4.1}$$

$$S \rightarrow C: \quad M' = \{FGResponse, F_c, F_s\}, \mathrm{MAC}_{K_r}(M') \tag{4.2}$$

Theorem 4. *It is not computationally feasible for an adversary to forge a response from the storage device such that the client will accept an F_s that has not been sent by the storage device.*

Proof: The same argument by which we show that K_a is a derived from P_a in Theorem 1 also shows that K_r is derived from P_r. Because response keys are not shared among clients, K_r is a shared secret between the storage device and the client. (Note, the administrator also shares the secret, but she is implicitly trusted.) Since the client will check that F_c is in the response, an adversary would have to send a response of the form $M'' = \{FGResponse, F_c, F_s'\}, MAC_{K_r}(M'')$, where F_s' is a guarantee generated by the adversary.

Since F_c a nonce generated by the client, the storage device has never generated a message with a prefix $FGResponse, F_c$ and MACed it with K_r. So, the adversary would have to be able to compute $MAC_{K_r}(M'')$ which violates the definition of the MAC. □

After the $FGResponse$ message is received, the client and storage device will have a shared F_s on which to build our generic message protocol. Up to this point we have not taken into account whether the freshness guarantee is a timer or counter. This will be considered when analyzing the generic message protocol in the next section.

4.5.2 The Generic Message Protocol

After the initial freshness guarantee exchange, we can send the normal SCARED requests to the storage device using the generic message protocol. Generic message requests take the following form:

$$C \rightarrow S : M = \{Operation, data, F_c, F_s\}, P_a, P_r$$
$$\text{MAC}_{K_a \oplus K_r}(M)$$

P_a is the public data associated with the access key K_a and P_r is the public data associated with the response key K_r.

Theorem 5. *On receipt of the generic request, the device can verify with overwhelming probability that the message came from a client in possession of K_a and K_r.*

Proof: The device can reproduce K_a and K_r from the P_a and P_r included in the request because it is in possession of K_d from which both keys are derived, so it can validate that the MAC is correct. K_a and K_r are secrets because they are from K_d using a pseudo-random function. Since K_a and K_r are secret, it would not be computationally feasible for an adversary that is not in possession of K_a and K_r to compute the MAC. Therefore, the message must have been sent by a client in possession of K_a and K_r. □

Theorem 6. *On receipt of the generic request, the device can verify with overwhelming probability that the message came from a client whose access is encoded in P_a.*

Proof: By Theorem 5 the device can verify the client possesses K_a. A client in possession of K_a has the access encoded in P_a by Theorem 1 or Theorem 3, depending on the derivation depth of K_a. □

Of course the device must also protect against replays of requests. An adversary can easily replay past messages which will have valid MACs.

Theorem 7. *With overwhelming probability an adversary cannot replay a request without detection by the storage device.*

Proof: Each time a request is issued to a device, the client must include a freshness guarantee. This freshness guarantee takes the form of a counter or a timer. To prove this theorem, we will examine each case.

Case 1: Freshness guarantee using a counter.

Device counters have the following properties: they are globally unique, and they are

incremented by the client with each request. Globally unique means that the device will issue freshness guarantees to clients in such a way that no client will ever receive or generate a number that another client received or generated. In our implementation, the initial freshness exchange, F_s, will be initialized to a 128-bit counter with a 64-bit nonce, based on the time, as the high 64-bits and zero as the low 64-bits.

Each time the device receives a request, it compares the counter, C, with the counter, C', that it received in the previous request. If $C \neq C' + 1$, it is considered invalid. Since no previous message will have the value C, no previous message exists to be replayed.

Case 2: Freshness guarantee using a timer.

When using a timer, the device initially gives out the value of its current timer to the client. The client synchronizes its own timer with the device, and uses what it believes to be the current device timer as the freshness guarantee in each request.

Because of network latency and clock drift, the device allows the freshness guarantee to be within a few seconds of its timer before considering the message invalid. If the replay occurs outside of this window, the device will detect it when the freshness guarantee of the message is compared with the freshness guarantee of the device. If the replay occurs within the window, the device is able to detect the replay by maintaining a list of all the MACs used within the window. □

When the client receives a response, it must be able to verify that the response was to a request that it issued and from the device to which it was issued. We will first prove the origin of the message and then prove that the client is able to verify that it came in response to its request. The general form of the response is as follows:

$$S \rightarrow C : M = \{Response, data, F_c, F_s\}, \mathrm{MAC}_{K_r}(M)$$

Theorem 8. *When a client receives a response, it is able to verify with overwhelming probability that the response came from the device to which it sent a request.*

Proof: K_r is received from the administrator by the client over the trusted channel. The device is able to produce K_r from P_r since it possesses K_d, but an adversary cannot. □

When proving Theorem 9, it is interesting to note that it holds even though the nonce need not be random. Since an adversary can predict the value of F_c, we need to use K_r in the request, as well as the response.

Theorem 9. *When a client receives a response, it is able to verify with overwhelming probability that it is in response to a particular request.*

Proof: F_c is a nonce that is generated by a client when it accesses the disk. This implies that no two requests generated by a client will have the same F_c. Theorem 8 allows the client to verify that the response came from the storage device. An adversary cannot cause the storage device to generate a message MACed with K_r since the device will only MAC a response using K_r if the request had a valid triple (P_a, P_r, K), where $K = K_a \oplus K_r$ and M is MACed with K. So by Theorem 5, a valid response can only contain an F_c from a previous request from the client. By Theorem 7, a replay of a previous request will be detected by the storage device. A replay by the adversary of previous responses from the device would have an old F_c that the client would reject. □

It should be noted that there is a security hole in the cache update call-back. Although we can detect replays of the cache update call-back, we can not detect if an adversary blocks a cache update message. Being able to block a cache update message is much more powerful that being able to replay an update message. A replay results in a degradation of file system performance since it will cause unnecessary reads, but will not cause invalid data to be used. When an update is blocked,

stale and invalid data will not be invalidated from the client cache, so the adversary could cause the client to use old data. This is an artifact of the caching protocol. A stronger consistency protocol such as that employed by DFS would avoid this problem, but would introduce others, such as denial of service from clients refusing to release cache tokens.

4.6 Summary

In this section we have presented a protocol to provide integrity, identity, and freshness guarantees to both the client and storage device. The protocol will work in both session based transport such as TCP or an non session based transport such as UDP. Even though we use timers in the non session based case, We do not require globally synchronized clocks.

Because of the derivation scheme presented in the previous chapter, not only is key exchanges between the storage devices avoided, we do not require that the administrator be able to communicate with the storage device. This allows greater freedom in the network topology.

Because of the few requirements we make of the network, clients, and storage device, and the security guarantees we provide, SCARED makes a good foundation for a distributed file system. In the next chapter we introduce a file system that builds on the SCARED object model to manage files and directories and the key derivation and security protocol to authenticate accesses to the storage device. Because of the services provided by SCARED we are able to avoid the need for a file server.

Chapter 5

Using SCARED in a Distributed File System

Using SCARED with network attached storage, we have the building blocks for a serverless distributed file system, called Brave. The file system is serverless in the sense that it does not need a file server to manage the meta data. Instead, the responsibility for managing the meta data is shared between the clients and the storage devices. The integrity of the file system itself is the responsibility of the clients.

It is useful to contrast Brave with other distributed file systems and network attached storage. On one extreme are NFS and CIFS which manages the complete file system on one server. On the other extreme are SAN storage devices which are basically SCSI devices connected to a network. A slightly less extreme example of a distributed file system is AFS. AFS has volume servers which manage a subtree of the distributed file system. On the other hand, a less extreme example of network attached storage is the NASD project which uses a file server to manage meta data for the file system, and allows the storage devices to store file data in objects managed by the

<table>
<tr><td rowspan="2">Entry tag</td><td rowspan="2">Lookup tag</td><td rowspan="2">Version</td><td rowspan="2">Filename</td><td rowspan="2">Location type</td><td>OID</td><td>Hostname</td></tr>
<tr><td colspan="2">Symlink</td></tr>
</table>

Figure 5.1: Brave directory entry layout.

devices.

Brave falls exactly between these extremes. Like NASD, Brave stores the file data in objects managed by the SCARED storage devices, but it also stores the file system meta data. However, it does not store a whole directory subtree like AFS does. The SCARED devices manage the file system data and meta data in objects they manage, but the clients implement Brave in their VFS layer to actually associate the objects stored on the SCARED devices into a file system.

In the following section, we will explain the semantics of the Brave file system and its method of operation. To provide a basis for these explanations, the next section explains how the data and meta data is organized in the file system. Section 5.2 will explain the semantics of the file system, and §5.3 will explain the way the various file system operations are carried out in Brave.

5.1 Brave File System Layout

Since we are building a file system on top of SCARED devices, there is a natural mapping of file system structures to SCARED structures: each directory is stored in a meta data object and each file is stored in a data object. In our initial implementation we maintain this one-to-one mapping. In the future, files and directories may be striped across data and meta data objects to improve performance and scaling. Mirroring may also be used to improve performance and reliability. Even with striping and mirroring, the basic concepts presented here remain unchanged.

To introduce Brave, we must first introduce the mount point or root of the file system. The

root of the Brave file system is a directory. In Brave there is not anything special about the root, any meta data object can be used as the root of the file system. Figure 5.1 shows the data structures that are stored in the entry data. In addition to these structures, the hash of the file name is stored in the lookup tag.

By storing the hash of the file name in the lookup tag, the SCARED disk is able to return the desired entry on a lookup without sending all the directory entries to the client. This saves network bandwidth, as well as optimizing one of the most common directory operations. Using a hash of the file name instead of the file name itself, the storage device is able to search on a fixed size number and preserves the secrecy of the file name, if needed.

Because the lookup tag is the hash of the file name and not the file name itself, the file name of the directory entry needs to be stored in the meta data entry. A SCARED device does not use the file name since lookups are done on a lookup tag, so the file name is stored in the entry data, which is stored without being interpreted by the storage device.

The main purpose of a directory entry is to provide a mapping between a name and a file or directory. For this reason, a pointer to the location of the file or directory follows the file name. The pointer comes in two forms: a symbolic link or an object identifier and host name (hard link).

A hard link is composed of the host name of the SCARED device that stores the object containing the file or directory and the object identifier (OID) of that object. A hard link preserves referential integrity. This means that an object referenced by a hard link will not be deleted until the link is deleted.

Unlike a hard link, a symbolic link does not retain referential integrity. Instead, the symbolic link is a string that is passed back to the operating system to be resolved. The string need not reference a file that is part of the Brave file system or even a file that exists.

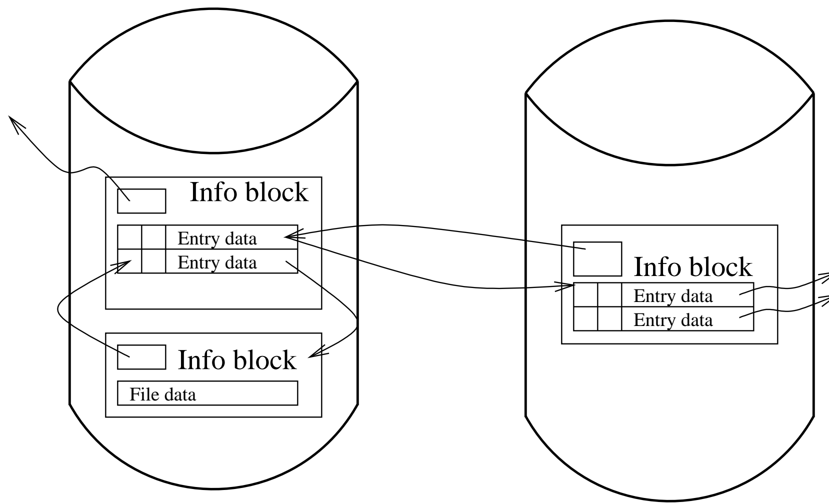

Figure 5.2: An example directory structure stored in a meta data object.

Referential integrity is preserved by using the info block that is part of every SCARED object. The info block is stored uninterpreted by the SCARED device. The Brave clients store the entry tag, the OID, and the host name of the meta data entry that references the object. Using the entry data and the info block, clients have pointers between the entry and object and *vice-versa*.

Figure 5.2 shows an example of a directory containing a file and directory. The file is stored on the same SCARED device as the directory, but the directory is stored on a different device. It is important to note that the SCARED device stores the info block, the entry data, and the file data, but does not use the contents. The figure shows both the forward links from the entries to the objects they reference, and the backward links from the objects to the entries that reference them.

5.2 Brave Semantics

As mentioned in the introduction, Brave extends local file system semantics to network storage. However, when other clients also have access to the network storage, there are additional semantics that arise. Network and client failure conditions also add semantics not present in local file systems. We present Brave semantics by first presenting the file semantics in §5.2.1 and then presenting the directory semantics in §5.2.2.

5.2.1 File Semantics

In a local file system, a file is presented as a logically contiguous stream of bytes. Files can grow and shrink, and bytes can be read and written at random locations. Files must be opened before they are accessed and closed after access is complete.

To improve write performance of files, UNIX uses a write back cache for files. File changes can be in the cache for up to one minute before it is actually written to non-volatile storage. An application can also immediately commit changes to non-volatile storage using the *sync* system call. Even though changes may not be committed to non-volatile storage, the local cache manager of a local file system reflects the changes to other applications on the local machine.

In a distributed file system there is a cache manager in each network client, so any uncommitted changes can be reflected locally at the client where the change occurred, but not in the other clients. This problem is further complicated by the fact that we do not require clients to be able to communicate with each other. For these reasons we write back changes to the SCARED devices when the file is closed, in addition to the normal commit process using the one minute timer and the *sync* system call. This is similar to the caching policy in AFS [24]; however, like JetFile [21] we only dispatch notifications to other clients when write back occurs instead of waiting for

the notifications to be received. Note, it is possible to use a caching policy that is closer to UNIX semantics by using the caching policy of Sprite [39], but we chose to implement our caching policy because of its simplicity and recoverability. This decision may be reevaluated later.

5.2.2 Directory Semantics

Directories have more structure than files. They store a set of entries indexed by a file name. Directory operations are always written through to non-volatile storage. We maintain these semantics in Brave. Just as with files, directory cache updates are dispatched to clients after changes are committed and do not wait for clients to acknowledge the updates.

UNIX allows multiple *hard links* to a file. Hard links allow a file to be referenced by multiple directory entries. UNIX limits hard links to files and only allows links to files on the same file system as the directory containing the link. Brave only allows a single hard link to a file. This simplifies the file system consistency checks.

To allow links to files on other file systems and to directories, UNIX also has *symbolic links*. While hard links preserve referential integrity, symbolic links may not point to a file or directory that actually exists and will not be updated if the object they point to is deleted or renamed. Brave supports symbolic links.

5.3 Brave Operations

The SCARED devices handle the management of the storage of the meta data and data objects, so most of the file system operations map directly to SCARED object operations. However, to implement all of the semantics the client must manage the relationship between the directory entries and the objects to which they point. The file operations correspond exactly to the SCARED

data object operations, so they will not be reviewed here. Instead, we will review the steps required to implement the directory operations. The storage devices treat meta data and data differently, which allows the SCARED devices to store the file system meta data without actually maintaining it or using it. The clients are in charge of keeping the meta data consistent and updated.

Brave maintains consistency of its directories. Since there is no central file server and no inter-disk and inter-client communication, special techniques are used to maintain referential integrity of the meta data across storage devices. These techniques are back links, ordered entry creation and deletion, and test-and-set updates.

Back links are used to check the consistency of the files system. In the info block of every object is the location and id object and the creation tag of the meta data that refers to the object. Consistency can be checked by insuring that objects referenced by meta data have a link back to that meta data, and meta data indexed by the back link of an object actually contains a reference to that object. The former case indicates the entry is invalid and should be deleted, and the latter indicates the object should be deleted.

5.3.1 Creation

Entry creation is a particularly troublesome operation in terms of referential integrity, since meta data stored in the directory entries must be sychronized with the info blocks of the objects to which the entries refer. To cope with client and disk crashes that may occur in the entry creation process, we have a well defined order of operations. First, the meta data is added to the meta data object. Next, the object is created with a back link to the meta data. Finally, the meta data is updated with the lookup tag, the name, and the location of the new object.

If the client or disk fail after the first step, the meta data will be cleaned up when the back

link check is done since the meta data does not refer to any object. Failure on the second step will result in the object being cleaned up, since the meta data does not reference the new object.

To avoid having to lock meta data objects in order to update the meta data, test and set updates are used. If part of the meta data is to be updated by a client, the old meta data must be read, updated, and rewritten. Since another client could update the meta data in the middle of the first client's update, the first client should be able to detect this condition. This is done by allowing a client to send a version tag of the old meta data. If the disk receives an update for meta data and a version tag that does not correspond to the current meta data, the update will be rejected.

5.3.2 Deletion

Deletion could be a much simpler operation than creation were it not for directory deletion semantics. Normal file systems require a directory to be empty before it can be deleted. Because of the distributed nature of Brave, we must synchronize the deletion of directory entries with the object itself to ensure that a directory that is empty at the start of the deletion operation, remains empty for the duration of the operation. We must also ensure that a client failure does not leave the file system in an inconsistent state.

For these reasons, we use a three phase delete. The first phase sets a bit in the entry data to indicate that a deletion operation is in progress for that entry. The next phase actually deletes the object. Finally, the entry is deleted.

By putting the pending delete bit in the entry data, we avoid having to add semantics to SCARED while still insuring consistency in the presence of client failures. Whenever a client encounters a directory entry with a pending bit set, it knows that it must check that the object is present before using the entry. So, if a client doing a deletion fails after the first phase of the deletion,

the other clients will have to do an extra transaction to check the existence of the referenced object, but the file system will still be consistent. If the client fails after the second phase, the other clients will detect the invalid entry since the object will no longer be present and will ignore the entry.

5.3.3 File System Checks

Although the file system remains consistent in the face of client failures, performance is adversely affected if many invalid or empty entries are present. For this reason it is important to periodically run file system checks.

The storage device only stores the file system data. It is oblivious to the relationship of the objects that it stores. So the file system checks must be done a client. In practice it will probably be the administrator that checks an individual storage device, however anyone with the appropriate authorization may do the check.

A check is done by simply sweeping the meta data objects on a disk and checking that its directory entries reference reference existing objects. A second sweep of all objects on the disk is needed to insure that each object has a backward pointer in it's info block that points to a directory entry with a forward pointer to the object. Any entries or objects that fail the sweep are simply deleted.

Fortunately, no global locks need to be obtained before doing this kind of check, so the checks can be done while the disk is serving data to other clients. The meta data object that is being fixed does not even need to be locked since only empty or deleted entries will be removed and therefore not affect the other clients. This means that the disks do not need to be taken off line or clients denied access to any part of the file system while the check is running.

5.4 Conclusion

By building on SCARED, Brave can be implemented completely at the clients, while providing consistent file system semantics to the users. By taking advantage of the object management facilities of SCARED, Brave avoids requiring clients to be able to communicate with each other.

The structure of the file system allows directories and files to reside on any network disk, thus enabling a high degree of scaling both in terms of storage size, as well as network bandwidth. At the same time, referential integrity of the directories is preserved using ordered updates and the objects themselves as synchronization points.

Because of the distributed nature of Brave, it would be unfortunate if global locks had to be obtained or access denied to specific objects in order to clean up performance degrades left by failed clients. Because of the semantics of SCARED, no locking at all needs to be done for file system checks. Since SCARED manages the entries themselves, a file system check running in the background can easily delete empty entries and entries pending deletion, without affecting the accesses of other clients.

Brave illustrates the power of object semantics at the network storage. Not only do we gain the scalability benefits of a serverless file system, but we also have the strong access control and authentication provided by SCARED.

Chapter 6

Implementing SCARED and Brave

To validate our file system design and network attached storage model we implemented SCARED and Brave in both Java and C. The C version of the client was done in the form of a Linux Virtual File System (VFS). In the course of our implementation, we not only validated our model, but also were able to get an idea as to the complexity introduced into the clients and storage devices.

Brave uses TCP [45] because of its good performance in a variety of environments. It allows us to operate well in LAN environments, as well as WAN environments, such as the Internet. Even though TCP is a stateful protocol, we can gracefully recover from TCP disconnects by transparently reinitiating the sessions when needed. This allows us to avoid the overhead of maintaining sessions that are not in use, and to recover from network storage reboots.

Although the necessary concepts to implement SCARED and Brave have already been introduced, it is necessary to review some of them in terms of current UNIX file systems to fully understand the complexities introduced by SCARED and Brave, and how they integrate with the VFS.

6.1 UNIX file systems

The Berkeley Fast File system (FFS) is the quintessential UNIX file system [34] which improved upon the original UNIX file system [48]. Most local file systems resemble it, and even those that differ greatly in their design, end up adapting to its structure. This adaption is mandated by the VFS interface in UNIX.

The two basic concepts that FFS is built on are i-nodes and directories. The i-node manages the storage allocation and the directories manage the name space.

6.1.1 I-nodes

The basic storage management unit in FFS is the i-node. The i-node contains ownership and access information, in addition to the locations of the disk blocks that store the i-node data. Because the i-node is stored on a single disk block, there is a limit to the number of data blocks that can be referenced in the i-node itself. FFS makes use of indirect blocks to allow files larger than the number of blocks than can be referenced by the i-node itself.

An indirect block stores a list of the locations of data blocks making up a file. They can also point to other indirect blocks for even larger files. One of the improvements of FFS over the traditional UNIX file system was to increase the size of the disk blocks, which reduced the number of indirect blocks needed while improving disk I/O by acting on larger chunks of storage. They also made an effort to allocate blocks in the same file close together.

6.1.2 Directories

Directories are stored in i-node and play a special role in the file system. Files are also stored in i-nodes, however, the data contained in the file data blocks are read and written using the

file system, but the data itself is not actually used by the file system.

The data blocks for directories are used by the file system. As mentioned earlier, directories manage the name space of the file system. In FFS, the directories store the names of the files and subdirectories of their children along with an i-node number in the data blocks. When a lookup on a name is done, the directory maps the name to an i-node number. The file system keeps i-node tables to map i-node numbers to the appropriate disk blocks that store the i-node.

Applications cannot read and write the raw data blocks of a directory i-node. Instead directories have operations to add, change, remove, lookup, and read directory entries. The file system translates the directory operations into operations on the data blocks.

6.1.3 Virtual File Systems

When NFS was implemented on UNIX, it became apparent that there was a need for an interface that would allow file systems writers to expose their file system to the kernel. To define the interface, it was necessary to choose an abstract file system model that the file systems would implement.

The interface is called the VFS [28] and the abstract model is patterned after FFS using i-nodes and directories. Specifically, each file or directory is represented by an i-node. The types of operations on the i-node depend on whether a file or directory is represented. For files, the basic operations are read, write, and trunc. The basic operations for directories are lookup, create, delete, rename, and read directory.

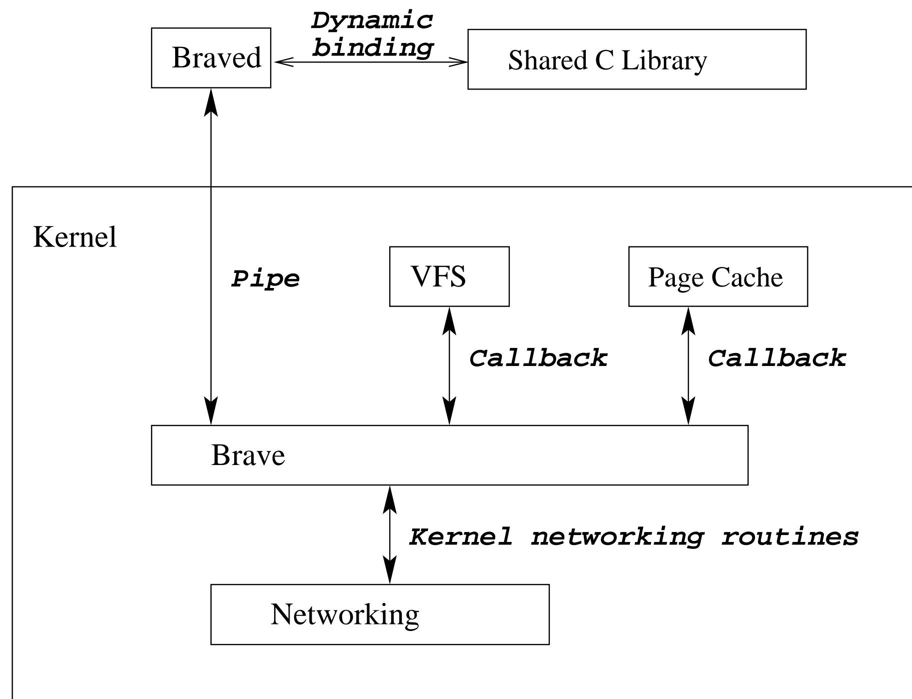

Figure 6.1: Brave integration into the Linux VFS.

6.2 Integrating Brave and SCARED into the VFS

Having introduced the VFS, it is useful to revisit briefly the new object paradigm introduced by SCARED. Normally, a disk exposes a block interface, so the file system must manage the mapping of blocks to i-nodes. This means that the file system must keep track of free lists, as well as maintain the block locations in the i-nodes and indirect blocks.

By using object based network storage, we move the management of the disk blocks and i-nodes to the storage device. Brave maps the VFS i-node operations directly to operations on the SCARED objects. This offloads the block allocation and management tasks from the client to the network storage.

As mentioned in the previous chapter, some of the meta data operations require additional work by the client file system to preserve the integrity of Brave. In our implementation this turned

out to be surprisingly easy. The biggest complication was creating an i-node from directory entries.

In FFS a directory entry simply mapped a name to an i-node number. A SCARED directory entry maps a name to a location. There is no concept of an i-node number in either SCARED or Brave. When a lookup is done and an i-node must be created, Brave extracts the information on the location of the object from the directory entry and instantiates an i-node placing the location information in a private file system specific member of the i-node structure. The i-node number is ignored by Brave.

When the i-node is used, Brave will need to establish a connection to the network storage. Since the host name of the network storage is stored in the directory entry, Brave will need to resolve the host name to an IP address in order to make the network connection. Brave uses a user level daemon, *braved*, to resolve the names and create the connection. This is because rules and functions to do host name resolution are in the shared C library, which is not loaded by the kernel.

Figure 6.1 shows links between Brave, the other parts of the kernel, and *braved.* When the Brave module is inserted into the kernel, it creates a new process and executes the *braved* program. Before starting execution it makes the standard input and output streams of the new process end points of pipes. The Brave kernel module and *braved* then communicate over these pipe.

When a connection needs to be established, the Brave kernel module will send a message through the pipe to *braved* containing the host name of the device to which the connection is to be made. The *braved* program uses the Posix *gethostbyname* routine to resolve the IP address and then open a TCP connection to that host. Either the open socket number or a -1, if unsuccessful, will be returned to the kernel through the pipe. If the connection succeeds, the Brave kernel modules will find the socket indexed by the socket descriptor in the file descriptor table for *braved* and use that socket to communicate with the network storage. A similar process is used to close the TCP

connection, as well as, obtain the keys to make SCARED requests.

The final piece that links Brave into the Linux kernel is the page cache. Brave takes advantage of the Linux page cache so that it only needs to handle reads of pages that are not already in the cache and writes. Whenever one of these conditions occur, a callback is issued by the page cache on the Brave i-node. The read and write functions are then directly mapped to SCARED requests to fulfill the request.

6.2.1 Allocation Management

Brave allows files and directories to reside on different SCARED devices than their parent directories. So when a new file or directory is created, the VFS needs to decide which device to use. If the same device is always used, the performance of Brave will be that of a single device. Other considerations also need to be taken into account, such as where the new file or directory is to be used, network locality of accesses, the free capacity of available devices, the estimated load that the new file or directory will generate, and the current load of available devices.

We chose to use a simple allocation policy that would allow behavior that could be well understood by users of Brave and allow the variables mentioned above to be taken into account. It should be noted that many of the above variables are really only known by the user at allocation time.

Our allocation policy is very simple, unless indicated otherwise by the user, a new file or directory is created on the same device as its parent. A user may indicate that the new object should be created on a different device using a special syntax. When the new file name contains a substring of the form "@{host}", the VFS allocates the new object on the device that corresponds to the given hostname. The special substring is removed before it is inserted into the directory. For

example the file “foo@{host.domain}” will be created as a file with the name “foo” on the device that corresponds to “host.domain”.

6.3 Implementing SCARED

The SCARED object model was designed to be as simple as possible, so we were surprised at the complexity of implementing the SCARED server. SCARED does not need to maintain a file system hierarchy or worry about referential integrity, so it is simpler than a normal local file system. However, it must maintain the information that is normally contained in an i-node as well as the list of free blocks on the disk. Thus, the object based disk is significantly more complex than a disk that simply needs to map block requests to sections of a disk.

Our implementation of the SCARED device bears strong resemblance to FFS. An object identity is mapped to a disk block that contains the info block and ACL for the object, as well as pointers to the data blocks or indirect blocks if needed. The device must also maintain a list of free blocks just like FFS. To provide fast recovery from power failures and reboots, we journal the meta data request.

Even though the code is not much simpler than a local file system, SCARED does have a shorter code path when doing lookups of objects. This is because the object identifier is used to directly index the disk block that contains the object’s data block locations. This allows us to skip the directory searching and the iterative lookups that are needed in the local file system. (These lookups take place in Brave at the client.)

It should be noted that we could have adopted a much simpler implementation similar to the Bullet file system, that would have resulted in much smaller code; but would have left us with a defragmentation problem as well as requirement that the clients be able to cache entire files at

locally. Fortunately, the object abstraction allows for changing the implementation of the storage management and allocation.

6.4 Summary

In summary, we were able to do a proof of concept of Brave and SCARED by implementing a Brave file system at the client and a SCARED server in both Java and C. The semantics of SCARED mapped well into the operations that Brave require.

When implementing the Linux VFS, the only difficulty was the resolution of host names by the kernel. The use of host names instead of IP addresses allows a level of indirection that eases the movement of network storage to different subnets. Because name resolution is highly configurable, we use a user space daemon to do the resolutions using the C shared library.

The C implementation of SCARED illustrated the complexity that is introduced by moving from a block interface to an object model. Even though we have adopted a very simple object model, we have to do the same management of free space and storage management that a local file system must do. The flat name space and object identifiers does reduce the amount of code in the implementation and the code path at runtime when doing meta data operations such as lookups.

Chapter 7

Conclusions

As clients becomes more connected, it becomes imperative to have a scalable and secure method of accessing network attached storage. We have reviewed the current methods of providing distributed file services and the common ways of securing them. We have identified the problems with these current methods and proposed ways to overcome their deficiencies.

7.1 Contributions

By building upon the SCARED object model and the SCARED authentication protocol, we have been able to implement an authenticated serverless file system, Brave. Because Brave is implemented at the client and does not require a central file server, we remove the limitations to scalability that file servers bring.

7.1.1 Comparison to Related Work

The most popular distributed file systems, NFS and CIFS, suffer from security, as well as scalability, problems. Both systems are inappropriate for usage on untrusted networks because

of the simplicity of compromising their security systems and both require the file system to reside entirely on a single server.

Clustering and NASD are two ways of increasing the scalability of the single server. A cluster that has a cluster file system, such as the serverless file system, can export an NFS file system from each node of the cluster; which allows the cluster to handle many more clients and export more storage than a single server, but it is still limited by the size of the cluster.

In similar ways, NASD allows the server to grow by offloading the file data server function to the storage devices. It has better security properties that allow confidentiality and integrity guarantees between clients and storage devices, but it still relies on the file server to serve the meta data and generate capability keys, which limits the scalability of the file system. NASD also requires modification of the clients so that the clients are able to direct file data requests to the storage devices instead of the server. NASD does not use identity keys which increases the number of keys managed at the client and obtained from the file server. Other advantages of SCARED over NASD are shared access keys and more efficient freshness guarantees when using session based protocols.

In many ways, the Brave file system is closest to AFS. It has the unified name space of AFS which allows it to scale by distributing the file system over multiple servers. Brave has three main advantages over AFS. First, SCARED devices do not mandate a specific security infrastructure, instead they can be incorporated into the existing security infrastructure. Second, AFS manages file system trees in terms of volumes. Each volume server maintains the file system hierarchy for that volume. In effect, AFS transparently mounts these file systems to achieve a unified name space. SCARED manages only individual files and directories. This means that SCARED's allocation is much finer grained than AFS. It also allows files to be striped across multiple disks, something that cannot be done with AFS. Finally, the clients manage where objects are allocated.

Normally, AFS requires an administrator to manage a volume group, but Brave allows clients to put files and directories on any storage device they choose. This not only gives the clients greater freedom, but also eliminates a management task for administrators.

7.1.2 Specific advantages of SCARED and Brave

Brave and SCARED have their own unique advantages. Brave offers scalability in terms of the amount of storage, as well as the number of clients. Brave's strong authentication is the result of the strong authentication of SCARED. The object model used by SCARED is powerful enough to eliminate the need for a file server and still provide authenticated access to storage. Finally, the SCARED protocol allows for simplified key management while making few requirements on the network topology.

The client directed allocation provides two kinds of scalability. First, as the need for storage increases, new SCARED devices can be added to the network. The addition of the device is independent of any central authority, so the aggregation of storage can grow without bound. Second, the allocation decisions are made at the client. This means scaling is not limited by management overhead. Each client manages its own storage, so as the number of clients and storage grows, so does the file system.

Brave and SCARED also offer strong authentication guarantees, allowing us to have more security than any of the available file systems. The ability to control access to the disk is dependent upon the SCARED object model. The model gives a control point for doing access control. Without an object model, access is done on a block basis. Usually, as in the case of Fibre Channel, the disk is also divided into partitions. Without the object model, access control can only be conveniently done on a disk or partition basis. The only other point of access control is the disk block. Unfortunately,

the disk blocks are small enough that to do access control on individual blocks would require a lot of work to convey to the disk which group of blocks a client is allowed to access.

With the object model, the disk blocks are grouped into objects which serve as an ideal point of access control. We are able to attach access control lists to the objects to allow easier key management at both the clients and storage devices.

The ability to provide the correct granularity of access control is reason enough to use the SCARED object model, but there are even more advantages in terms of object allocation and management. By allowing the disk to manage the allocation of disk blocks, many of the synchronization issues associated with managing allocations can be “centrally” at the disk itself. In addition, the disk can also be used as a synchronization point for meta data operations, such as file creation and deletion.

Along with the object model, we have presented and analyzed a method of off-line shared key derivation and an authenticated network protocol. The key derivation avoids the computation overhead of public key operations and the infrastructure requirements of other symmetric key authentication methods. The authentication protocol provides identity, integrity, and freshness guarantees, without requiring the use of heavy cryptographic operations or encryption.

We have validated both SCARED and Brave by implementing a SCARED server and Brave client in the form of a Linux VFS. The implementation validated the simplicity of the Brave client when used with SCARED. It also illustrated the increased complexity of a storage device, when it needs to do more than just direct mapping of requests to disk blocks.

In conclusion, SCARED is a flexible object model and security protocol which can be used in a variety of environments. When used with Brave, the combination results in a file system that works well in both a LAN and WAN environment, making it perfect for use on the Internet

because of its scaling and security properties. The small footprint of Brave allows it to be used in a small device, as well as a large server.

7.2 Future Work

The purpose of the current work was to establish a basis for building a distributed file system based on authenticated network attached storage. The current design is robust enough to allow for additional semantics to be added to the SCARED model and new schemes for mapping file system files and directories onto the SCARED devices. Specifically semantics may be added to enable different forms of caching. Locking is also missing from Brave and may be supported by extending SCARED. Allocation and load balancing could aid in the performance scaling of Brave. Striping and mirror could improve the performance of large files and provide redundancy for performance and reliability reasons.

7.2.1 Caching

We have a simple notion of cache coherency that allows us to have a caching model similar to AFS without a lot of overhead at the storage device. In our current implementation of the Brave client, we do only in-memory caches. Depending on the network bandwidth and latency between the client and storage device it may be more efficient to have a large on-disk cache at the client, as is used with AFS.

It is also possible to simplify the storage device by using a time based caching policy like NFS. This would eliminate the need to track objects the clients are interested in, as well as the need to send cache call backs. On the other extreme, a more strict cache coherency protocol, like the one used in DFS, can be used. This would require more state at the storage device, as well as more

communications between the clients and storage device.

Because of the distributed nature of Brave, we believe the best policy would be to allow clients and storage devices to negotiate the caching policy on a per device, or even per object, basis. This would allow for a wide variety of clients and storage devices.

7.2.2 Locking

We have not addressed locking. Distributed file systems vary on their support of locking. NFS has a locking protocol that is used with the file sharing protocol. CIFS has strict locking built into it. AFS does not support file locking.

It is our belief that locking is best done outside the file system. However, locking semantics can be added to objects or to directory entries. Since files may be striped across multiple objects, doing the locking on the directory entry would allow for centralized management of the locks for the set of objects that constitute the file or directory.

Another approach to locking would be to simply use a separate locking service. Information on which service to use could be encoded into the directory entry. While this would allow locking to be done without having to add semantics to SCARED, work would need to be done to insure that the authorizations for the locking service and SCARED objects that correspond to the locks are synchronized.

7.2.3 Striping and Mirroring

Our current implementation of Brave has a one-to-one mapping between a file or directory and a SCARED object. Greater performance can be obtained by striping files and directories across SCARED objects. Replicating files and directories over multiple objects allows client access even

in the presence of network and device failures. Replication also increases the number of clients that are able to access a given file or directory.

The format of the directory entry data allows for the location of a file or subdirectory to be in a variety of formats. Currently, the only types of locations are symbolic links and pointers to single object. More complicated locations, such as a list of objects the file or directory is replicated across, or a list of objects and a stride size for striping, can be stored in the entry data to allow a variety of striping and mirroring schemes.

The difficulty managing a file or directory that is stored on multiple objects is the coordination of the updates. When only one object is involved, the device managing the object also serves as a point of synchronization. When more than one object is involved, we no longer have a synchronization point.

If locking semantics were available to the Brave clients, they would be able to coordinate updates to the objects; although recovery from client failures would still need to be addressed. It would be nicer to be able to provide the ability to do the necessary coordination without full locking semantics.

7.2.4 Allocation and Load Balancing

Currently, we allow the users complete control over where new objects are allocated when a file or directory is created. This type of allocation is useful because a user may have a better idea of how, where, and when a file will be used than a file server could possibly have. As files are used, a file server, if there were one, would be able to detect access patterns and hot spots, move files to localize client access, and spread hot spots across devices.

The problem with this kind of load balancing and allocation management is the lack of

a central server. Potentially, a device could gather local information on object accesses and an allocation manager could gather access statistics to make more global allocation decisions.

Appendix A

Key Data Encoding

SCARED key data is made up of a set of attributes that correspond to a key. Each attribute is encoded by a byte representing the attribute type, a byte representing the length of the attribute data, followed by the attribute data. Using this encoding at most 254 attributed types can be encoded since the types zero and 255 are reserved. Also, the attribute data can be at most 255 bytes in length.

The data that corresponds to a SCARED key is made up of a set of attributes that correspond to a key appended to the attributes of the keys from which it was derived. Since the keys can be derived from a number of other keys, the attributes of the parent keys need to be delimited. We delimit each set using the octet 0xff. Thus, key data will be composed of sets of attributes delimited by the octet 0xff, where the first set of attributes are the attributes associated with the first key from which all of the subsequent keys are derived.

Table A lists the defined attribute types. These attributes fall into three categories: identity, capability, and informational. The attributes in each of these categories will be described in the following sections. Section A.4 describes the algorithm for evaluating the attributes to check access.

octet	type
0x01	client id
0x02	object id
0x03	permission
0xfd	expiration
0xfe	salt

Table A.1: SCARED attribute types for key data.

A.1 Identity Attribute

The identity attribute conveys the identity of the possessor of the key. This identity could take the form of a 16-bit UID or GID, or a variable length string, or any other sequence of bytes identifying a client. In our current implementation we have chosen to have the user identifiers restricted to 128-bit numbers. This allows for globally unique identifiers to be generated and used. Thus each identity attribute will have 16 bytes of attribute data associated with it.

While it is conceivable that a key could have only one identity associated with it, in our applications we associate multiple identities with a key. A user will generally have her 128-bit identity in the key data, as well as the 128-bit group ids of the groups to which she belongs. Each identity will be a separated attribute in the key data.

When evaluating the key, the identities in an attribute set will be treated as a union. This means that adding an identity to an attribute set will broaden the access a key has. When deriving a key that contains an identity attribute, the derived key will only have a subset of the access of the parent key. This means that a key, K' that is derived from a key, K with the identity attributes for A, B, and C, and has the identity attributes for B and D in the key data for K', will only identify the possessor of K' as B. Even though D is in the key data for K', it is ignored since it is not in the key data for K.

Permission	Mask
read	0x0001
write	0x0002
delete	0x0004
admin	0x0008

Table A.2: Permission masks for the permission capability attribute.

A.2 Capability Attributes

The octets 0x02 and 0x03 are capability attributes. They describe what the key can do as opposed to who possesses the key. The octet 0x02 is the object capability and restricts the capability to a specific object, and the octet 0x03 is the permission capability and describes the permissions of the capability.

The object capability binds the capability to a specific object. It is always 16 bytes in length and the attribute data will contain the OID of the object. If there are multiple objects attributes in an attribute set, the capability will apply to both objects. As with identity attributes, a derived key may include object capabilities to further restrict the objects to which a capability applies, but they cannot increase the number of objects to which a capability applies.

By itself, the object capability attribute does not carry any permissions, so unless it is derived from a key with permission attributes or an identity attribute, the key would not be able to do anything. A permission capability attribute gives permission to the holder of a key. The permission attribute is usually 16-bits in size. It is a bit mask, whose interpretation is given in table A.2. Only the first set of attributes can add a permission to a key. Permission capabilities in any of the other attribute sets will only further restrict the permissions of a key. If the permission attribute is not accompanied by an identity or object capability attribute and is not derived from a key with an identity or object capability attribute, the permission applies to all the objects on a storage device.

So a key with just the read permission capability attribute would be able to read all objects on the storage device.

It should be noted that capability attributes and identity attributes can be mixed. For example, if the read capability attribute occurs with the identity attribute for B, the key is able to read all objects that B can read. Another example would be an object capability attribute for object I and the identity attribute for B. This key would be able to read I if B could read I, but it would not be able to read J even if B could read J.

A.3 Key Information Attributes

There are two attributes that have information about the key itself and are orthogonal to the capability and identity aspects of the key. The octet 0xfd has expiration information about the key, and octet 0xfe describes the *salt* used to derive the key. Both of these attributes enhance the security aspects of the key by providing a way to limit the lifetime of the key and to randomize the generation of the key.

Limiting the lifetime of the key limits the window of opportunity for an attacker to use a compromised key. The expiration time is relative to the local timer on the storage since we do not require a global clock. The timer is a 64-bit big-endian number and the units are seconds.

A *salt* is a number that is added to the derivation of the key to introduce randomness. The salt itself need not be random. Salts are usually 128-bit numbers. The storage device does not use the salt for anything.

Since response keys must be unique but do not carry any access rights, their key data is only made up of salt. For example, a response key will usually take the form of the octet 0xfe followed by the length of the salt, usually 16, then 16 arbitrary bytes. The salt does not have to be

random but must be unique in relation to a network storage device.

A.4 Key Data Evaluation

To determine the rights and validity of a key, the storage device must evaluate the sets of attributes in the key data starting with the first set of attributes. The first step in attribute evaluation is to check for an expiration attribute. If one is present and expired, the key will be rejected as expired.

The next step is to check for an identity attribute. If one is present the access list for the target of the request is checked to insure that the operation is permitted.

Assuming that the access list permits the operation, the third step is to check that the OID of the target is in one of the object capability attributes. If there are object capability attributes present and the target of the operation is not in one of the capabilities, the request will be rejected.

The fourth and final step is to check for the permission attribute. If there is no identity attribute and there is no permission attribute and the first attribute set is being evaluated, the key is rejected as invalid. If a permission attribute is present, the key is rejected if one of the permission attributes does not allow the requested operation.

The steps are repeated for each set of attributes in the attribute set. If the key is not requested in any of the passes, the operation is permitted by the key data. Before the operation is actually carried out, the freshness guarantees must still be checked and the key corresponding to the key data must be generated and the MAC checked.

Appendix B

Pseudo-Random Functions

The following was taken from the lecture notes [14] of Cynthia Dwork's Foundations of Cryptography class at Stanford. These notes draw heavily from [37, 19, 33].

> A truly random function $f : \{0,1\}^n \longrightarrow \{0,1\}^n$ has no short (polynomial in n-sized) representation. Intuitively, a pseudo-random function $g = g_s : \{0,1\}^n \longrightarrow \{0,1\}^n$, specified by a short (say, $O(n)$-bit) seed, is polynomial-time indistinguishable from a truly random function, in that a polynomial-time bounded adversary, querying a function h at adaptively chosen points $x \in \{0,1\}^n$, cannot determine whether h is pseudo-random or truly random.
>
> ...
>
> **Notation.** Let $\mathbb{N}$ denote the set of all natural numbers. Let I^n denote the set of all n-bit strings, $\{0,1\}^n$. Let U_n denote the random variable uniformly distributed over I^n.
>
> The following definitions are taken from [37]. See also [19, 33].
>
> Informally, a pseudo-random function ensemble is an efficient distribution of functions that cannot be efficiently distinguished from the uniform distribution. That is, an efficient algorithm that gets a function as a black box cannot tell (with non-negligible success probability) from which of the distributions it was sampled. To formalize this, we first define function ensembles and efficient function ensembles:
>
> **Definition 4 (function ensemble).** *Let ℓ and k be any two $\mathbb{N} \mapsto \mathbb{N}$ functions. An $I^\ell \mapsto I^k$* function ensemble *is a sequence $F = \{F_n\}_{n\in\mathbb{N}}$ of random variables, such that the random variable F_n assumes values in the set of $I^{\ell(n)} \mapsto I^{k(n)}$ functions. The* uniform $I^\ell \mapsto I^k$ function ensemble*, $R = \{R_n\}_{n\in\mathbb{N}}$, has R_n uniformly distributed over the set of $I^{\ell(n)} \mapsto I^{k(n)}$ functions.*

Definition 5 (efficiently computable function ensemble).
A function ensemble, $F = \{F_n\}_{n\in\mathbb{N}}$*, is* efficiently computable *if there exist probabilistic polynomial-time algorithms,* $\mathcal{I}$ *and* $\mathcal{V}$*, and a mapping from strings to functions,* ϕ*, such that* $\phi(\mathcal{I}(1^n))$ *and* F_n *are identically distributed and* $\mathcal{V}(i, x) = (\phi(i))(x)$*.*

We denote by f_i *the function assigned to* i *(i.e.* $f_i \stackrel{\text{def}}{=} \phi(i)$*). We refer to* i *as the key of* f_i *and to* $\mathcal{I}$ *as the key-generating algorithm of* F*.*

For simplicity, we concentrate on the definition of pseudo-random functions and on their construction on length-preserving functions. The distinguisher, in our setting, is defined to be an oracle machine that can make queries to a length-preserving function (which is either sampled from the pseudo-random function ensemble or from the uniform function ensemble). We assume that on input 1^n the oracle machine makes only n-bit queries. For any probabilistic oracle machine, $\mathcal{M}$, and any $I^n \mapsto I^n$ function, O, we denote by $\mathcal{M}^O(1^n)$ the distribution of $\mathcal{M}$'s output on input 1^n and with access to O.

Definition 6 (efficiently computable pseudo-random function ensemble). *An efficiently computable* $I^n \mapsto I^n$ *function ensemble,* $F = \{F_n\}_{n\in\mathbb{N}}$*, is* pseudo-random *if for every probabilistic polynomial-time oracle machine* $\mathcal{M}$*, every polynomial* $p(\cdot)$*, and all sufficiently large* n*'s*

$$\left|\Pr\left[\mathcal{M}^{F_n}(1^n) = 1\right] - \Pr\left[\mathcal{M}^{R_n}(1^n) = 1\right]\right| < \frac{1}{p(n)}$$

where $R = \{R_n\}_{n\in\mathbb{N}}$ *is the uniform* $I^n \mapsto I^n$ *function ensemble.*

In this thesis we use the term "pseudo-random functions" as an abbreviation for "efficiently computable pseudo-random function ensemble". We also refer to the key, i, of f_i as the secret used with the pseudo-random function.

Bibliography

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in *Proceedings of the 13th Symposium on Operating Systems*, pp. 198–212, ACM, October 1991.

[2] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology – Crypto 96 Proceedings*, pp. 1–15, 1996.

[3] M. Bellovin and M. Merritt, "Limitations of the Kerberos authentication system," in *Proceedings of the Winter 1991 USENIX Conference*, pp. 253–267, January 1991.

[4] S. M. Bellovin, "Security problems in the TCP/IP protocol suite," *Computer Communication Review*, vol. 19, pp. 32–48, April 1989.

[5] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung, "The KryptoKnight family of light-weight protocols for authentication and key distribution," *IEEE/ACM Transactions on Networking*, vol. 3, pp. 31–41, February 1995.

[6] M. Blaze, "A cryptographic file system for UNIX," in *First ACM Conference on Communication and Computing Security*, pp. 9–16, November 1993.

[7] R. C. Burns, R. M. Rees, and D. D. E. Long, "Safe caching in a distributed file system for

network attached storage," in *International Parallel and Distributed Processing Symposium*, May 2000.

[8] B. Callaghan, B. Pawlowski, and P. Staubach, "NFS version 3 protocol specification." RFC 1813, June 1995.

[9] J. P. Chandler, D. C. Arrington, D. R. Berkelhammer, and W. L. Gill, *Identification and Analysis of Foreign Laws and Regulations Pertaining to the Use of Commercial Encryption Products for Voice and Data Communications.* National Intellectual Property Lay Institute, George Washington University, Washington, D.C., January 1994.

[10] J. S. Chase, D. C. Anderson, A. J. Gallatin, A. R. Lebeck, and K. G. Yocum, "Network I/O with trapeze," in *HOT Interconnects*, IEEE, August 1999.

[11] M. Dahlin, *Serverless Network File Systems.* PhD thesis, University of California at Berkeley, 1995.

[12] T. Dierks and C. Allen, "The TLS protocol version 1.0." RFC 2246, January 1999.

[13] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, pp. 644–654, November 1976.

[14] C. Dwork, "The non-malleability lectures." http://Theory.Stanford.EDU/˜gdurf/cs359-s99/notes1a.ps, 1999. Excerpt quoted in appendix B.

[15] C. Everhart, "Security enhancements for DCE DFS." OSF RFC 90.0, February 1996.

[16] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for lightweight sessions and application level framing," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 784–803, December 1997.

[17] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, “A cost-effective, high-bandwidth storage architecture,” in *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 92–103, October 1998.

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

[19] O. Goldreich, S. Goldwasser, and S. Micali, “How to construct random functions,” *Journal of the Association for Computing Machinery*, vol. 3, no. 4, pp. 792–807, 1986.

[20] O. Goldreich, *Foundations of Cryptography (Fragments of a Book).* Weizmann Institute of Science, February 1995.

[21] B. Grönvalli, A. Westerlund, and S. Pink, “The design of a multicast-based distributed file system,” in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 251–264, USENIX, September 1999.

[22] J. H. Hartman and J. K. Ousterhout, “The Zebra striped network file system,” in *Proceedings of the 14th Symposium on Operating Systems Principles*, pp. 29–43, ACM, December 1993.

[23] D. Hitz, J. Lau, and M. Malcom, “File system design for an NFS file server appliance,” in *USENIX San Francisco 1994 Winter Conference*, pp. 235–246, January 1994.

[24] 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,” in *ACM Transactions on Computer Systems*, vol. 6.1, pp. 51–81, February 1988.

[25] IBM, *IBM General Parallel File System for AIX: Installation and Administration Guide*, second ed., October 1998.

[26] R. H. Katz, “High performance network and channel-based storage,” Tech. Rep. UCB/CSD 91/650, University of California at Berkeley, Sept. 1991.

[27] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas, “DEcorum file system architectural overview,” in *Proceedings of the Usenix Summer 1990 Technical Conference*, (Berkeley, CA, USA), pp. 151–164, Usenix Association, June 1990.

[28] S. R. Kleiman, “Vnodes: An architecture for multiple file system types in Sun UNIX,” in *USENIX summer conference*, pp. 238–247, USENIX, 1986.

[29] J. T. Kohl and B. C. Neuman, “The Kerberos network authentication service.” RFC 1510, September 1993.

[30] H. Krawczk, M. Bellare, and R. Canetti, “HMAC: Keyed-hashing for message authentication.” RFC 2104, February 1997.

[31] E. K. Lee and C. A. Thekkath, “Petal: Distributed virtual disks,” in *7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-96)*, pp. 84–92, October 1996.

[32] D. D. E. Long, B. R. Montague, and L.-F. Cabrera, “Swift/RAID: A distributed RAID system,” *Computing Systems*, vol. 7, pp. 333–359, Summer 1994.

[33] M. Luby, *Pseudorandomness and Cryptographic Applications.* Princeton University Press, 1996.

[34] M. K. McKusic, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, pp. 181–197, August 1984.

[35] R. Merkle, "Secure communication over insecure channels," *Communications of the ACM*, vol. 21, no. 4, pp. 294–299, 1978.

[36] P. V. Mockapetris, "Domain names – concepts and facilities." RFC 1034, November 1987.

[37] M. Naor and O. Reingold, "Synthesizers and their application to the parallel construction of pseudo-random functions," *JCSS: Journal of Computer and System Sciences*, vol. 58, 1999.

[38] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM*, vol. 21, pp. 993–999, dec 1978.

[39] B. Nelson, B. Welch, and J. Ousterhout, "Caching in the sprite network file system," *ACM Transactions on Computer Systems*, pp. 134–154, January 1988.

[40] C. Neumann and T. Ts'o, "Kerberos: An authentication service for computer networks," *IEEE Communications Magazine*, September 1994.

[41] The Open Group, *Protocols for X/Open PC Internetworking: SMB, Version 2*, September 1992.

[42] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 15–24, December 1985.

[43] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proceedings of the 10th Symposium on Operating System Principles*, pp. 15–24, December 1985.

[44] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” in *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pp. 109–116, June 1988.

[45] J. Postel, “RFC 793: Transmission control protocol.” RFC 793, September 1981.

[46] B. Preneel, A. Bosselaers, R. Govaerts, and J. Vandewalle, “A chosen text attack on the modified cryptographic checksum algorithm of cohen and huang,” in *Advances in Cryptology – Crypto 89 Proceedings*, pp. 154–163, 1989.

[47] E. Riedel and G. Gibson, “Understanding customer dissatisfaction with underutilized distributed file servers,” in *Proceedings of the Fifth NASA Goddard Space Flight Center Conference*, September 1996.

[48] D. M. Ritchie and K. Thompson, “The UNIX time-sharing system,” *Communications of the ACM*, vol. 17, pp. 365–375, July 1974.

[49] R. Rivest, “The MD5 message-digest algorithm.” RFC1321, April 1992.

[50] R. L. Rivest, A. Shamir, and L. M. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, pp. 120–126, February 1978.

[51] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” in *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 1–15, ACM, October 1991.

[52] M. Sach, A. Leff, and D. Sevigny, “LAN and I/O convergence: A survey of the issues,” in *IEEE Computer*, pp. 24–32, December 1994.

[53] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network file system," in *USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer*, pp. 119–130, 1985.

[54] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A scalable distributed file system," in *2nd Symposium on Operating Systems Design and Implementation (OSDI-96)*, pp. 224–237, USENIX, October 1997.

[55] U.S. Department of State, "International traffic in arms regulations (ITAR)." 22 CFR 120-130, Office of Munitions Control, November 1989.

[56] U.S. Department of State, "Defense trade regulations." 22 CFR 120-130, Office of Defense Trade Controls, May 1992.

[57] U.S. Government, "Proposed federal information processing standard for digital signature standard (DSS)." Federal Register, August 1991.

[58] U.S. Government, "Proposed federal information processing standard for secure hash standard." Federal Register, January 1992.

[59] R. van Rennesse, A. S. Tanenbaum, and A. Wilschut, "The design of a high-performance file server," in *Proceedings of the 9th International Conference on Distributed Computing Systems, IEEE*, pp. 22–27, June 1989.

[60] M. Wittle and B. Keith, "LADDIS: The next generation in NFS file server benchmarking," in *USENIX Summer Conference*, pp. 111–128, USENIX, June 1993.