Security for a High Performance Commodity Storage Subsystem

Howard Gobioff

July 1999
CMU-CS-99-160

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the degree Doctor of Philosophy

This research is sponsored in part by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-002. Additional support was provided by the member companies of the Parallel Data Consortium, an NSF Graduate Research Fellowship, and the United States Postal Service. The views and conclusions in this document are my own and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

Keywords: computer security, network attached storage, network attached secure disks, NASD, capability system

Abstract

How do we incorporate security into a high performance commodity storage subsystem? Technology trends and the increasing importance of I/O bound workloads are driving the development of commodity network attached storage devices which deliver both increased functionality and increased performance to end-users. In the network attached world, storage devices co-exist on the network with their clients, application filemanagers, and malicious adversaries who seek to bypass system security policies. As storage devices move from behind the protection of a server and become first-class network entities in their own right, they must become actively involved in protecting themselves from network attacks. They must do this while cooperating with higher level applications, such as distributed file systems or database systems, to enforce the application's security policies over storage resources. In this dissertation, I address this problem by proposing a cryptographic capability system which enables application filemanagers to asynchronously make policy decisions while the commodity storage devices synchronously enforce these decisions.

This dissertation analyzes a variety of access control schemata that exist in current distributed storage systems. Motivated by the analysis, I propose a basic cryptographic capability system that is flexible enough to efficiently meet the requirements of many distributed storage systems. Next, I explore how a variety of different mechanisms for describing a set of NASD objects can be used to improve the basic capability system. The result is a new design based on remote execution techniques. The new design places more access control processing at the drive in order to deliver increased performance and functional advantages. Based on the performance limitations of software cryptography demonstrated in a prototype implementation of a network attached storage device, I propose and evaluate an alternative to standard message authentication codes. This allows storage devices to precompute some security information and reduces the amount of request-time computation required to protect the integrity of read operations. Finally, I discuss the availability of cryptographic hardware, how much is required for a network attached storage device, and the implications of adding tamper-resistant hardware to a storage device.

Dedicated to the memory of John D. Gannon,
a great teacher and mentor to many

Acknowledgments

There are several people without whom this work would not have been possible, first and foremost are my advisors, Garth Gibson and Doug Tygar. They gave me the opportunity to explore my two primary research interests at the same time: systems and computer security. Many years of collaboration with both of them have come to fruition in this dissertation. Despite being neither my advisor nor a member of my thesis committee, David Nagle gave me large amounts of his time and served as a virtual third advisor for my final year or two at Carnegie Mellon University.

My doctoral work is also a direct result of collaboration with the marvelous people of the Parallel Data Lab (PDL). Khalil Amiri, Fay Chang, Joan Digney, Jennifer Landefeld, Patty Mackiewicz, Paul Mazaitis, Erik Riedel, David Rochberg, and Jim Zelenka all contributed directly to the success of this work. I am indebted to Khalil and Erik for their friendship, support, and feedback that helped me get through a graduate school career that was filled with ample bumps and potholes. I'm the first to venture beyond CMU but soon they will follow me into the real world. Fay and David helped with the original NASD work as well as being there for the duration while we all diverged into our separate research agendas. Jim Zelenka was instrumental in getting the NASD prototype systems on which I did most of my research up and running . Last but not least, Joan, Jennifer, Patty, and Paul all contributed to a friendly environment and helped protect me from the administrivia of the university and the mundane details of keeping the research group running so that I could do my job — research.

Of course, the atmosphere and people in the School of Computer Science were an important part of my graduate school experience. The opportunity to learn about a wide variety of topics through an abundant supply of seminars and hallway discussions has helped me learn more than simple classroom teaching could ever achieve. For this, I will forever me thankful. I would also like to thank all my friends in SCS who both helped me stay sane during six years of graduate school and with whom I have shared the graduate school journey.

I would also like to thank the Parallel Data Consortium (PDC) members who provided useful feedback on how to balance the interesting technical challenges versus real world concerns through engaging discussions during their visits to CMU and the PDC retreats. They provided me with insight and understanding into the concerns faced in designing a commodity storage device. Specifcally, I'd like to thank Daniel Stodolsky and Hugo Patterson at Quantum as well as Dave Anderson and Chris Malakapalli at Seagate for useful discussions on the inner workings of drive electronics.

Finally, I would like to thank my parents, Sharon and Bruce, who gave me the freedom to develop my intellectuall curiosity throughout my entire life. Never placing unwarranted limits on what I chose to do, they simply prevented me from doing too much damage to myself or the world. I would also like to thank my brother Neil for his constant support during my years at CMU.

— Howard Gobioff
Pittsburgh, PA
7/30/99

Table of Contents

List of Figures

Chapter 1: Introduction

Network attached storage devices are a reality. As the storage industry tries to deliver improved performance to the end user from commodity storage subsystems, it has moved towards adopting technologies that attach low-cost storage devices directly to the network. But despite the fact that the industry is moving in this direction [NSIC96, Seagate98, SNIA98], there has been little research done on the security issues of network attached storage and the optimization of the security system for overall system performance. My dissertation describes the design and implementation of a secure system efficiently supporting access control, integrity, and privacy in a network attached storage system.

When a computer system is used by a single individual and all hardware is maintained in an environment with controlled access, the user can have a high level of confidence in the system's integrity. But in a distributed system environment, new concerns arise: system components are stored in physically disparate locations and components communicate over a potentially insecure communications network. As a result, a distributed system is open to attacks over the network, such as forging messages on the network, tampering with the messages' contents, replaying or reordering messages, spoofing a user's identity, or denying service to valid requests.

With increasingly abundant resources on the network, security is becoming a greater concern both in academia and in industry, as witnessed by the widespread coverage of any security problems in the popular press. In the last fifteen years, the pervasive model of computing has gone from personal computers, terminals, and mainframes to networked resources across a department, building, campus, or the world. As the usage of distributed resources has increased, the security community has developed a variety of techniques and algorithms to address security concerns [Schneir96, Menezes98] in applications such as electronic commerce [Cox95, Yee95], authentication [Neuman94], and function shipping [Gosling96, Necula96].

Distributed filesystems such as the Andrew File System (AFS) [Howard88, Satyanarayanan89], Network File System (NFS) [Sandberg85, Callaghan95], Secure File System (SFS) [Mazieres97], Serverless Network File System [Anderson96c], Sprite [Ousterhout88] and Truffles [Reiher93], separate components of the filesystem for performance, management, and data sharing. The older of these, such as NFS and Sprite,

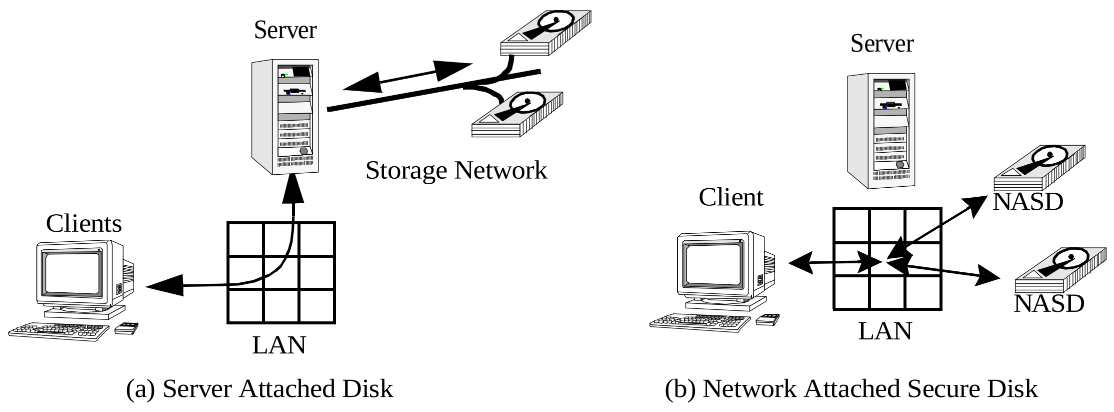


Figure 1-1 SAD versus NASD

On the left, the traditional approach, which I call Server Attached Disks (SAD), forces the server machine to copy data from a storage network onto a LAN. On the right, the Network Attached Secure Disk (NASD) architecture removes the server from the datapath and allows clients to directly transfer data with the storage devices.

assume clients are trusted and pay little attention to security while AFS, SFS, and Truffles are unwilling to make this assumption and, instead, assume that clients may be compromised or even malicious and therefore place a greater emphasis on system security.

The Network Attached Secure Disk (NASD) architecture pushes the distributing system components a step further. A traditional distributed filesystem (Figure 1-1a) aggregates all the data behind one, or a few, servers whose primary tasks are supporting high-level filesystem semantics and copying data from a storage network, such as SCSI, to a general-purpose commodity network, such as Gigabit Ethernet. In contrast, the NASD architecture (Figure 1-1b) allows all communication to occur over a single general purpose network and removes the fileserver from the datapath, significantly reducing the server's workload and increasing its scalability [Gibson97a]. Because the server is no longer limiting the storage-to-client datapath bandwidth by copying through the server CPU, the storage system can deliver greater cost-effective bandwidth to clients than is possible with more conventional approaches [Gibson98].

In network attached storage systems, storage devices themselves must play an active role in the security of the entire distributed system, rather than having fileservers perform this function. Because fileservers no longer protect the storage devices, a malicious party can directly address the devices. In extreme cases, the storage may be physically accessible and subject to direct attack.

A storage device participates as part of a distributed application (such as a distributed filesystem, database, or video server) and serves as a repository for data. A single type of storage device may be used in a wide variety of applications, thus only the most common set of semantics are fixed in the NASD interface [Gibson97b], a proposed network attached storage system. The limited interface of storage devices forces most of the application specific complexity, including the security policies, to reside in the filemanager, essentially a fileserver that does not move data. However, the enforcement of these policies must occur synchronously at the storage device. This requires that the NASD interface include a mechanism through which a filemanager can express policy decisions to the drive. These decisions must be enforced on each request without decreasing or eliminating the performance advantages of the NASD architecture.

The critical security requirement for network attached storage is that the storage device must protect the integrity of all data that users entrust to it. Specifically, the drive should verify that the filemanager has authorized each request and that no adversary has tampered with any request. Privacy, in contrast, is an optional requirement because it can be provided, with some limitations, at the application (e.g. filesystem) layer. This is not an option for integrity; an application cannot protect the integrity of the data unless the storage devices directly participate in this task. Without integrity support at the drive, an attack could easily modify message traffic to alter any data that was being read or written to storage and effectively destroy system integrity.

While many of the same technology trends which improve workstation and server performance also improve storage devices, storage devices fall far short of servers in terms of DRAM, computational capacity, and cost. These relative shortages thus make efficient resource utilization more critical in storage devices than in servers. Hard disk drives — the largest portion of the storage market— compete in a very cost sensitive market; thus it is important to add only the required functionality with minimal added resource requirements. The limited resources motivate a design that is focused on limiting both the amount of state and computation necessary to provide security.

These issues all coalesce into a need for a security subsystem that addresses, first, the requirement that a storage subsystem be suitable for different applications, and, second, storage devices' resource constraints. My thesis statement can be summarized as follows.

> ***A cryptographic capability system designed for a range of distributed storage applications provides fundamental scalability because it enables reuse of policy decisions and unmoderated, parallel interactions between application and device. Furthermore, commodity storage devices can be designed to inexpensively provide security and high bandwidth.***

To validate the thesis, I present a basic capability design that delivers synchronous enforcement of security policy with asynchronous involvement of the server. This separation of roles delivers the scalability advantages demonstrated in earlier work [Gibson97a]. I address the throughput limitations of software cryptography by careful selection of cryptographic primitives, a novel application of message digests to protect integrity, and cryptographic hardware. These options are evaluated by a combination of prototype implementation and simulation results. Finally, I address physical security by constructing a high-level design for a tamper-resistant network attached storage device.

Through the process of validating my thesis claim, this dissertation describes a design for such a system, then provides both an analysis of the design and an implementation of the design. It makes the following novel contributions:

- An argument for the separation of policy and mechanism in a commodity network attached storage system enforced by a cryptographic capability system,
- The basic design and implementation of a security system for network attached storage, based on cryptographic capabilities,
- An understanding of the scalability advantages of aggregation mechanisms that move more functionality to the storage device,
- A proposal to use precomputed hash values as the basis for a new message authentication code structure,
- A demonstration of the performance advantage of the new message authentication code structure,
- An understanding of the performance requirements for message authentication code computation,
- An evaluation of available options for hardware support, and
- A high-level sketch of a NASD design based on tamper-resistant hardware.

Chapter 2 presents some background on network attached storage and its motivating trends, a discussion of the security needs of network attached storage, and describes my basic model of a network attached storage device. Chapter 3 surveys the access control policies of a set of filesystems and databases to illustrate the diversity of requirements that may be placed on the NASD security system. Chapter 4 presents the basic design of the NASD security system based on capabilities, which I implemented in the CMU NASD prototype system, and analyzes the security of the system. Chapter 5 explores the implications of a variety of alternatives to capabilities from the perspective of the filemanager and the storage device. Chapter 6 describes an alternative message authentication code structure enabling the drive to precompute message digests and provide strong levels of security with reduced resource requirements at the drive. Chapter 7 explores a range of performance points for MAC bandwidth. Chapter 8 surveys current physical attacks against computing devices and discusses what is necessary to build a tamper-resistant network attached secure disk. Chapter 9 concludes and presents some future directions for further research.

Chapter 2: Background

In this chapter, I introduce the problem domain for my dissertation. This chapter introduces the idea of network attached storage and the security challenges that it creates. I start with a description of current storage systems and their performance constraints. Second, I describe the technology trends which enable network attached storage to improve over current storage systems. Third, I describe the network attached secure disk architecture, as well as my model for its storage devices. After introducing the basic research area, I discuss the security problem in the context of network attached storage and how it differs from past storage systems. Finally, I conclude with a discussion of work related to the basic network attached storage architecture.

2.1 Server Attached Disks: the Status Quo

The server attached disk (SAD) architecture, shown in Figure 2-1, is the storage architecture most familiar to office and local area networks where storage is privately connected to a server machine. In a SAD system, the server machine can bottleneck the entire storage system when the system is under heavy load. A significant portion of the server's resources are dedicated to simply moving data from a storage network to a local area network. In this section, I will explain both the role of the server and accepted techniques for improving storage system performance.

The server's purpose is to provide some *application* to a client using a set of storage devices. An application consists of a set of well defined behaviors that clients expect to see when accessing the storage system. Examples of applications that could be built on top of network attached storage are a web server, a distributed filesystem, or a database system. A principal measure of an application's cost is the computational power required from the server machine to service a group of clients [Howard88, Nelson88].

The server's primary task is to act as an interface for clients to a storage system. Clients issue requests over a local area network (LAN) and the application-specific server processes the request and then forwards the request directly to the storage devices. In turn, the storage devices generate a reply to the server which the server forwards to the clients. In this architecture, the server must first copy the request from the LAN to the storage network and then the reply from the storage network back to the LAN. In filesystems,

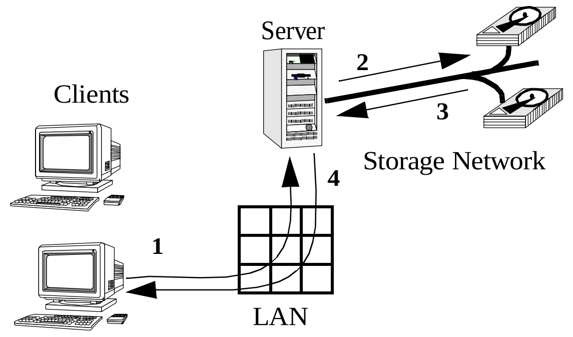


Figure 2-1 Server Attached Disks

Server attached disks are the familiar LAN distributed filesystem or database system. The server is responsible for receiving client requests over the LAN (1) and transforming the request into a storage operation (2). The storage devices then replies to the server (3) which forwards the replies to clients (4). A significant portion of the servers task is copying data from the storage network to the local area network.

simply moving the data between these two networks can account more than 50% of the server's load [Gibson97a].

Past research has demonstrated several well understood techniques to improve system performance, such as caching, striping, and closer integration. These form the basic toolbox for building a distributed storage system.

Client caching can reduce the server's load by satisfying requests at the client machine. For example, AFS clients use local disks to cache a portion of the distributed filesystem's files. Requests to cached data never leave the client machine. Since local caches absorb requests, it is possible to reduce server load. While client caching is essential for high performance, increasing file sizes and increased sharing are inducing more cache misses per cache block [Ousterhout85, Baker91].

Striping is another mechanism that improves the scalability of servers and I/O bandwidth. Individual filesystem objects are divided into stripe-units which are distributed, i.e. striped, across multiple fileservers. This enables parallel transfers of larger datasets while balancing the load over multiple servers [Cabrera91, Hartman93]. The large parallel transfers provide greater throughput to a client than a single server could provide to clients by spreading the request load across multiple underlying storage devices. The balancing reduces the "hot-spot" phenomena caused by the ad-hoc balancing of a filesystem's namespace, which is familiar from multiple-disk mainframe systems [Kim86]. Other research on striping systems has emphasized redundancy at the controller level [Cao93] and the management problems that come from incremental growth in storage systems that span multiple storage servers on the network [Lee96, Thekkath97].

Server integrated disks (SID), an approach closely related to SAD, exploits the fact that server machines are frequently dedicated resources which provide only a single service to clients. In a SID system, the server is built using a combination of highly optimized software and special-purpose hardware that is dedicated to a specific application. Thus, the server can service requests more efficiently than a general-purpose machine and operating system. These highly specialized systems have evolved to fill an important high-performance market niche [Hitz90, Hitz94]. Architecturally, SID systems are very similar to SAD systems but reflect special-purpose optimizations in their increased costs.

Server systems are an expensive approach to scaling storage bandwidth. With server integrated or server attached disks, placing a new disk on the server adds storage capacity but does not add bandwidth to the clients unless the server already has excess CPU and networking resources. Adding the CPUs and network interfaces necessary to deliver the bandwidth of the storage subsystem to the clients adds an overhead of about 80% for server resources to deliver the raw disk performance to clients [Gibson98].

The difficulty of scaling servers and aggregate bandwidth in a server based storage systems presents an opportunity to improve over the status quo with a network attached storage architecture, which is presented in Section 2.3. In the next section, I will describe some of the technology trends that enable us to adopt a network attached storage architecture and exploit this opportunity for improvement.

2.2 Motivating Technology Trends

Technology trends are enabling storage devices to directly deliver their performance to clients rather than requiring the assistance of an intervening server. In [Gibson98], the CMU NASD group argues that the role of the commodity storage device is changing as a result of the synergy of several technology trends: I/O bound applications, new drive attachment technologies, rapidly increasing drive performance, convergence of peripheral and interprocessor switch networks, and excess of on-drive transistors.

- **I/O-bound applications**: Traditional distributed filesystems workloads are dominated by small random access to small files whose sizes are growing with time, though not dramatically [Baker91, TPC98]. In contrast, new workloads are much more I/O-bound, including data types such as video and audio [Quantum99], and applications such as data mining of large data sets such as retail transactions, medical records, multimedia databases, or telecommunication call records.
- **Rapidly increasing drive capabilities**: The storage industry has been improving areal densities at 60% per year to help meet the increased application demands. The same technology improvements are also driving up disk bandwidth at 40% per year while driving down the cost per megabyte by 40% per year [Grochowski96].
- **New Drive attachment technologies**: The storage industry has evolved SCSI technology through a variety of similar technologies such Wide SCSI, UltraWide SCSI, and Fast Wide Differential SCSI to deliver rapidly increasing drive performance. The high transfer rates of modern drives has put pressure on the physical and electrical design of bus-based technologies such as SCSI to dramatically constrain the bus length (similar to the problems faced by Ethernet as it has evolved from a 10 Mbps standard to 100 Mbps and Gigabit/sec versions). For this reason, the storage industry is moving towards transporting SCSI communication over Fibrechannel [Benner96], a serial, switched, packet-base peripheral network that supports long cable lengths and high

bandwidth. For example, Quantum has publicly displayed their JINI demonstration disks which use an Ethernet interface [Wolfe99]. Attachment technologies that are more network oriented than a peripheral bus, such as Fibrechannel and Ethernet, also offer significantly greater addressability through name spaces that are orders of magnitude larger than a SCSI bus's 16 device limit.

- **Convergence of peripheral and interprocessor networks**: The amount of modern scalable computing research being done and the number of products based on clusters of commodity workstations are increasing. In contrast to the special-purpose interconnects of massively parallel computers such as the IBM's SP2, Intel's Paragon, and CalTech's Cosmic Cube, cluster computing typically uses standard protocols over commodity LAN routers and switches. To make clusters effective, low latency network protocols and user-level access to network adapters have been proposed [Wilkes92, Maeda93, Boden95, Horst95, vonEicken95]. Additionally, a new adapter card interface, the Virtual Interface (VI) Architecture has been standardized [Intel97]. Preliminary implementations of the VI Architecture in the Giganet GNN 1000 network adapter provide almost 100% of the available network bandwidth in a Gigabit/sec network, with less than 10% host-CPU utilization and extremely low latencies [Giganet98]. These developments continue to narrow the gap between the channel properties of peripheral interconnects and the network properties of client interconnects [Sachs94] and make the competing storage connection technologies — Fibrechannel and Gigabit Ethernet — look more alike than different. Recently proposed interconnect technologies for workstations such as the Next Generation I/O (NGIO) Forum indicate an industry move towards a switched technology for device attachment to workstations [NGIO99].
- **Excess of on-drive transistors**: The increasing transistor density in ASIC technology has enabled disk drive designers to lower cost and increase performance by integrating sophisticated special-purpose functional units into a small number of chips. Figure 2-2 shows the block diagram for the ASIC at the heart of Quantum's Trident drive. When drive ASIC technology advances from 0.68 micron CMOS to 0.35 micron CMOS, drive vendors could integrate a 200 MHz StrongARM microprocessor on to the same ASIC and still have the equivalent of approximately 100,000 gates of space for on-chip DRAM, cryptographic support, or network support while maintaining the same die size. For example, Siemens, Cirrus, and PalmChip all have products integrating RISC microcontrollers on ASICs with drive-specific functions [Siemens97, Cirrus99, Palmchip99].

These trends all increase demands on storage subsystems while enabling storage devices to provide higher level functions, compared to simple block oriented interfaces like SCSI, and deliver increased performance to clients over storage networks that are very similar to general-purpose networks. These trends point the way towards network attached storage systems which will address some of the performance shortcomings of server attached or sever integrated disks.

(a) Current Trident ASIC (74 mm^2 at 0.68 micron) **(b) Next-generation ASIC (0.35 micron technology)**

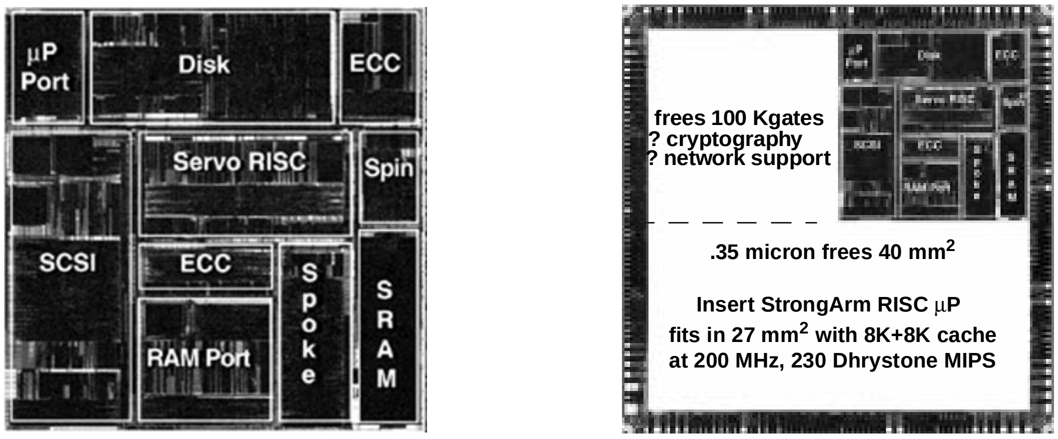


Figure 2-2 Quantum Trident ASIC

Quantum's Trident disk drive features the ASIC on the left (a). Integrated onto this chip are multiple functional units with a total of about 110,000 logic gates and a 3 KB SRAM: a disk formatter, a SCSI controller, ECC detection, ECC correction, spindle motor control, a servo signal processor and its SRAM, a servo data formatter (spoke), a DRAM controller, and a microprocessor port connected to an external RISC processor. By advancing to the next higher ASIC density (b), this same die area could also accommodate a 200 MHz StrongARM microcontroller and still have space left over for DRAM or additional functional units such as cryptographic or network accelerators.

2.3 Network Attached Secure Disks

Network attached secure disks (NASD), the storage architecture explored by Carnegie Mellon University's Parallel Data Lab (PDL) [Gibson97a, Gibson98], addresses the scalability, throughput, and cost issues of server attached storage by directly attaching storage to the network and bypassing the server on common operations as shown in Figure 2-3.

The NASD architecture changes the server's role from being actively involved in every request to a management role of providing high level application-specific semantics to clients. The server is no longer on the datapath and its responsibilities have changed so I refer to the remaining server functionality as the *filemanager*. The filemanager is responsible for defining policy with regard to who can access storage as well as for adding high-level functions such as cache consistency and namespace management. While I will refer to this vestigial server as the filemanager throughout the dissertation, a filemanager could be the management portion of any other application built on NASD such as a database.

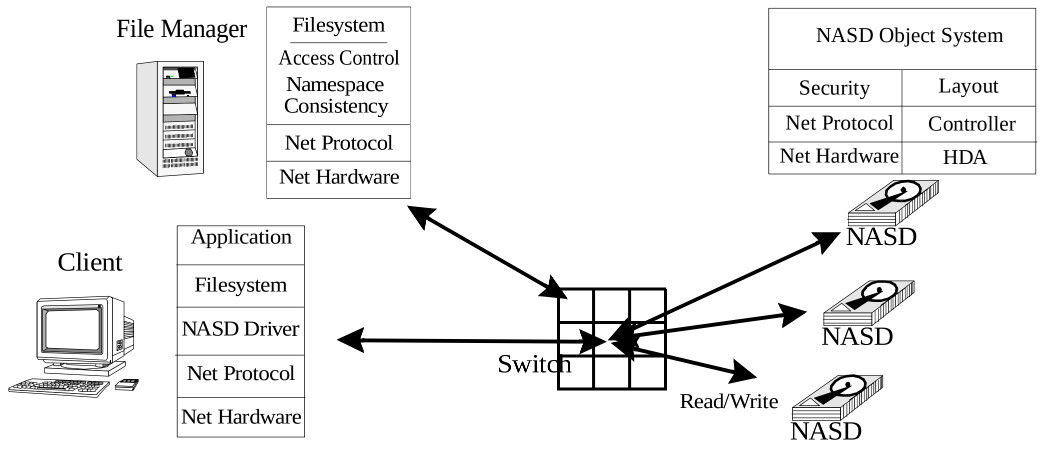


Figure 2-3 An Overview of the NASD Architecture

The major components are annotated with the layering of their logical components. Clients infrequently consult the filemanager when policy decisions are necessary, thus minimizing the load on the filemanager. However, for most read/write operations, the clients directly communicate with the storage devices eliminating the store-and-forward inherent through a server in more traditional storage architectures.

The NASD architecture can reduce the load on the server machine as well as increase the aggregate bandwidth delivered to clients. In previous work, CMU's NASD group demonstrated the NASD architecture can reduce the filemanager's load, during burst activity, by a factor of five for an AFS on NASD and by a factor of ten for an NFS on NASD system when compared to a server attached disk architecture [Gibson97a]. Additionally, CMU's NASD group demonstrated, in experiments using up to 8 client-disk pairs, that a NASD specialized filesystem can deliver linear scaling of bandwidth to clients [Gibson98].

The CMU NASD group argues that network attached storage should have the following properties[Gibson98]:

- **Direct transfer:** Data is transferred between the drive and the client without indirection or store-and-forward through a server machine.
- **Object-based interface:** Drives export variable length "objects" instead of fixed-sized blocks which gives drives direct knowledge of the relationships between disk blocks and minimizes security overhead. This feature also improves opportunities for storage self-management by extending into a disk an understanding of the relationships between blocks on a disk [Anderson98].
- **Asynchronous oversight:** This is the ability of the client to perform most operations without synchronous appeal to the filemanager for authorization. Frequently consulted but infrequently changed policy decisions, such as authorization decisions, are encoded into access credentials generated by the filemanager, given to clients, and subsequently enforced by the drives.

- **Cryptographic integrity:** By attaching storage to the network, the storage is now a first-class network entity and open to direct attack by adversaries. Thus, it is necessary to use cryptographic techniques to defend against potential attacks and enable storage to effectively enforce the policies of the fileservers.

While direct transfer and an object-based interface are simple properties, asynchronous oversight and cryptographic integrity pose greater challenges. Asynchronous oversight challenges the architecture to minimize the involvement of the fileserver in order to increase the filemanager's scalability. Asynchronous oversight and cryptographic integrity represent the challenge in how the filemanager enforces its policy over storage and how to protect network communication with the drive.

The focus of the NASD project has been on redefining the function and role of the most basic storage device, the hard disk, although much of the research is also relevant to other definitions of a storage device such as a RAID array or DVD jukebox. In order to allow arbitrary applications to be implemented on the NASD architecture, the architecture separates management and application-specific semantics from generic data movement operations. A sophisticated server machine handles the former while a low-level storage device focuses on the latter.

Our group has studied and proposed a design for a next generation interface for commodity storage devices [Gibson97b]. This interface is currently the basis for pre-standards discussions among storage vendors as a potential follow-on to SCSI as part of the National Storage Industry Consortium's Network Attached Storage Working Group [NSIC96] and formed the starting point for Seagate's Object Oriented Disk (OOD) draft specification [Seagate99b].

2.3.1 Drive Model

In this section, I introduce my working model of network attached storage as used in the remainder of the dissertation. I present a very basic description of the NASD interface, as well as some basic expectations about a drive's working environment and hardware. A more detailed discussion of the interface to the drives can be found in [Gibson97b].

A NASD disk divides its available storage capacity into a group of disjoint portions of capacity called *partitions*, which are defined by a set of *cryptographic keys* that enable administration by filemanagers (as discussed at length in Section 4.3) and by a portion of the drive's overall capacity. Similar to traditional partitions in a Unix or DOS system, the capacity of a partition is expected to change infrequently, but, unlike a traditional partition, the actual low-level storage blocks associated with the partition can be changed freely by the storage device because the block assignments are hidden below the interface that storage presents to the application.

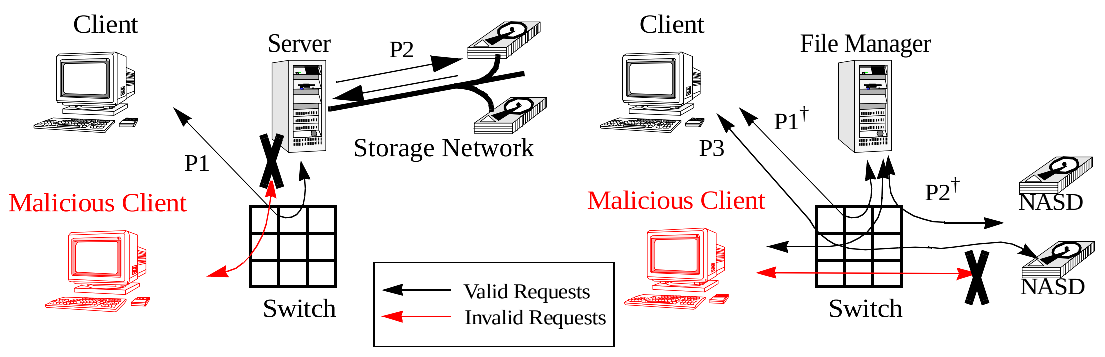
Within each partition, applications create file-like *objects* containing a single set of attributes, the metadata information exported by the drive, and a variable length sequence of bytes. In some applications, a NASD object will correspond to an entire file while other applications use an object to store a fragment of a file, such as a stripe unit for data striping, or database table. The access control system and management of the storage takes place at the object level in order for the drive to make local decisions about efficiently managing its storage below the interface.

In the previous section, I discussed how disk drive ASICs are rapidly gaining resources as they ride the same technology curves as workstation microprocessors. However, they are currently, and will remain for the foreseeable future, two or more generations of technology behind high performance processors. Because of the cost-sensitive commodity nature of storage, manufacturers are unable to incorporate leading edge semiconductor technology into a disk drive. Thus, they will always be resource poor relative to modern workstations.

Early NASD drives will have a processor comparable to a 200 MHz StrongARM, Intel's high performance embedded processor from the ARM processor family [Jaggar96], to perform most of the NASD control functions beyond low level media management functions which can be borrowed from existing drives. This class of embedded processor can readily be integrated onto a commodity drive ASIC when manufacturers move to 0.35 micron fabrication technology as shown in Figure 2-2 [Gibson98]. Past research by the CMU NASD group has concluded that a 200 MHz StrongARM (or equivalent) processor can handle the storage device's control task, but special communications implementations are necessary because current client-server RPC and networking systems can consume 70+% of CPU cycles when a system is performing high bandwidth data transfers [Gibson98].

Because of cost constraints, a NASD drive will have relatively small amounts of memory, perhaps 8 to 16 MB, a factor of 2 to 4 more than current drives. In contrast, a server-class machine has several hundred megabytes. Security, data caches, metadata caches, run-time stacks, and, potentially, remotely executing code [Riedel98a] will compete for memory. Additionally, only a small amount (currently 3KB which may increase by a factor of 2x-8x) is on the primary ASIC and thus readily available in the early stages of processing a request. The limited availability and contention from multiple sources for memory motivates a design that limits the amount of memory consumed for security functions.

I assume that a NASD drive will serve a large pool of users such as in an enterprise-wide filesystem application. In contrast, in modern Fibrechannel [Benner96], a storage networking technology, attached disks are designed to interoperate with a small set of server machines operating in a small area, using connection-based communication. In order for a drive to concurrently support a large number of users and to minimize fault tolerance concerns, the NASD architecture is based on RPC-style semantics where the drives do not maintain significant amounts of state, with the exception of the stored data, across requests.



(a) Server Attached Disks (b) Network Attached Secure Disks

Figure 2-4 SAD vs. NASD.

On the left in a Server Attached Disk (SAD) System, the server is responsible for all of the security properties of the system. A client makes a request to the fileserver via an application specific protocol (P1) and the fileserver transforms the request into a storage request in the storage network protocol (P2). If the fileserver does not approve the request, the request never goes beyond the server. On the right in the NASD case, the client uses a modified for NASD version of the application specific protocol (P1†) to request access to a NASD object. Later, the client uses the NASD client protocol (P3) to make direct requests to the drives. If the request is invalid, the drives are responsible for recognizing that the client is performing an unauthorized operation. When the filemanager needs to communicate with the drives, it uses a third protocol (P2†) that uses the NASD interface but allows the filemanager unrestricted access to the storage device. Together, P2† and P3 form the NASD protocol.

2.4 The Security Problem

This thesis addresses the core security problem in any storage system: How does an application enforce its security policy over its storage devices? In a server attached disk system, the application server machine is in a position to directly enforce its policies over storage. Any request that a client, malicious or benevolent, makes to storage must pass through the server machine. Before the server passes the request along to the storage device, the server can examine the contents of the request and make a decision whether to allow or disallow the request as shown in Figure 2-4a. By virtue of its location on the data path, the server is able to control who can access storage devices.

But the NASD architecture removes the server machine from the datapath thus the server can no longer synchronously enforce its policies by inspecting each and every request. A client is able to make requests directly to the storage device and the storage device, rather than the filemanager, must decide if the request is valid as shown in Figure 2-4b. This change forces drives to become actively involved in providing the application's security rather than passively accepting all requests. In NASD, the application has the same security goals as in a server attached system while the application server is not on the datapath. This requires that the security system address both of the security and performance goals.

In a server attached disk system, there are two protocols being used: an application specific client-server protocol (P1) and a generic server-storage protocol (P2). The client-server protocol includes whatever security properties the application implements. The server-storage protocol includes no provision for security.

The NASD architecture decouples the making of a policy decision from the enforcement of those decisions. This decoupling creates the basic structural change which forces storage to directly handle security. The two protocols in the SAD case, P1 and P2, expand to include a third generic client-storage protocol, P3, which clients use to directly access storage. However, the filemanager is ultimately responsible for its storage and its policy decisions; these must be enforced on any request made using the client-storage protocol without requiring the synchronous involvement of the filemanager i.e. asynchronous oversight. As I will discuss in Chapter 3, there are a variety of kinds of access control policies that an application may want to enforce over its storage devices. This prohibits fixing a single access control policy in the NASD interface and encourages a flexible and generic approach to the security mechanism. Instead, filemanagers essentially package up their access control decisions into an *access credential* and give it to the client through the NASD version of the client-filemanager protocol, $P1^{\dagger}$. The access credential is later used by the client in P3 to demonstrate its access rights to the drive. This mechanism which allows the filemanager to preauthorize client requests is presented in detail in Chapter 4.

Storage device communication protocols, $P2^{\dagger}$ and P3, combine to form the NASD interface while the $P1^{\dagger}$ protocol between the filemanager and client is part of the application. This is a critical difference because all applications will be built on top of the NASD interface — which will be fixed by the storage devices — while the filemanager-client protocol can be entirely redesigned for each application. The NASD interface must provide enough functionality and flexibility to allow an application to be efficiently implemented on the NASD subsystem as well as meeting the application's security needs.

In addition to authorizing client operations, the filemanager may also require the drive and client to cooperate to protect the integrity and/or privacy of communication. The goal of *integrity* is to ensure that information has not been altered by unauthorized or unknown means. Protecting the integrity of communication is required to allow the drive to enforce the filemanager's policies. Without integrity, the drive is unable to verify that data has arrived unaltered, consequently the drive can not verify the data's origin and without knowing the origin, the drive can not verify that the request was authorized by the filemanager. The goal of *privacy* is keep information secret from all but those who are authorized to see it. In contrast to integrity, privacy is not necessary for the filemanager to control access to the disk. It is only necessary if the application wants data to be private.

More concisely, the security problem in NASD is to:

- Enable applications to implement their application-specific policies over network attached storage devices.
- Protect the integrity of communication involving network attached storage.
- Deliver the scalability and aggregate bandwidth potential of the NASD architecture.
- Optionally, protect the privacy of communication involving network attached storage.

In addition to needing secure communications in a NASD system, users are implicitly placing trust both in the workstation they use to access storage and in the physical security of the storage devices.

A client workstation may be located in hostile environments, such as publicly accessible clusters, or other physically accessible location where an adversary can take control of a client workstation. Hence the NASD architecture, as well as any many other distributed systems, must assume that a client is untrusted until the client proves otherwise. An adversary can alter the client machine’s operating system to modify data or release data contrary to both the policies of the system and the will of an innocent user. For this reason, when a user accesses data using a workstation, she is implicitly placing trust in the workstation not to misuse any information the user provides to the workstation, such as file data, cryptographic keys, or access credentials.

In small environments, the network can be physically protected because it may be entirely contained within a secure facility. However, most systems will have some connection to the outside world which may occasionally be breached and will allow an adversary a path to the internal secure network. Additionally, any user community of a non-trivial size will inevitably have malicious or disruptive users against whom the system must protect itself. A system that relies entirely on the strength of a firewall and the goodwill of its user population will eventually be compromised. For example, unhappy employees may take advantage of their “inside” status to destroy critical data before being fired, or may modify personnel records for personal gain.

In some environments, the storage devices may not be in a physically secure environment, and thus require physical as well as communication security. In Chapter 7, I discuss the hardware functions necessary to address a storage device operating in an insecure environment.

2.5 NASD Related Work

In this section, I present work related to the basic idea of network attached storage and the NASD architecture. For more detailed discussion of the general related work, the reader is referred to [Gibson97a, Gibson98]. Specific related work pertinent to the major points of this dissertation will be included within appropriate chapters.

The scalability problems of a centralized file server are widely recognized. Companies such as Auspex and Network Appliance have attempted to increase file server performance through the use of special purpose server hardware and highly optimized software [Hitz90, Hitz94]. In contrast, NASD attempts to increase server scalability by simplifying the job of the server rather than optimizing the server for data movement thus allowing low-cost workstations to act as servers for high performance storage systems.

About a decade ago, the storage industry moved from a physical-geometry storage interface to the logical block-based interface defined by SCSI [ANSI86]. The indirection of the SCSI interface has enabled many transparent improvements in storage performance such as RAID, transparent failure recovery, real-time geometry-sensitive scheduling, buffer caching, read-ahead and write-behind, compression, dynamic mapping, and representation migration [Patterson88, Gibson92, Massiglia94, STK94, Wilkes95, Rummler91, Varma95]. By further raising the interface to an object level, NASD enables the storage device to locate logically-related disk blocks nearby (which requires information unavailable to a disk in a block interface) and transparently provide features such as copy-on-write semantics for a fast copy operation and compression.

High-bandwidth data transfers can be achieved by striping data across storage devices or servers [Gibson92, Hartman93, Drapeau94]. In order to deliver this performance to clients in a network environment, NASD requires that a switched network fabric rather than a single shared-media network be used. With a switched network, many of the ideas of striping data across servers or local disks can evolve into a network attached storage environment [Amiri99].

The idea of simple, disk-like, network-based storage servers as the basis for higher-level storage servers has been under exploration for many years [Birrel80, Katz92]. The Mass Storage System Reference Model (MSSRM) advocated the separation of control and data paths almost a decade ago [Miller88] and logically "DMA"s the data from the storage from to the client. This is an optimized version of a distributed filesystem built on a storage controller, such as RAID. In contrast, NASD does not move the data through any storage controller but rather directly moves data from our smarter storage devices to the client.

ISI's Netstation project proposes an alternative object-interface called Derived Virtual Devices (DVD) which is the most similar project to NASD and demands the greatest explanation [vanMeter96]. When a drive boots up, it first authenticates to Kerberos authentication system [Neuman94] and then requests its basic configuration information from a remote controller called a Network Virtual Device Manager (NVDM).

The filemanager then authenticates to Kerberos and tells the physical disk which of its blocks are to be allocated to a specific virtual disk, equivalent to a NASD object. Within this virtual disk, the clients and storage manager can build either a filesystem or a single file. By using a simple block-oriented interface, DVD requires that the filemanager maintain all of an object's metadata and the filemanager ships it to the disk when the metadata is necessary for a client access. This prevents a DVD from exploiting the extra knowledge that an object interface provides to the disk.

Since the filemanager must build filesystem objects out of disk blocks at request time, DVD adopts a stateful connection-based model so it can avoid repeatedly sending the object metadata per client operation while the connection is open. The drive also associates security information with connection state. For a large number of concurrent clients, this could cause performance concerns as well as being a fault tolerance issue. Furthermore, the multiple message round trips to define a filesystem object can significantly reduce client performance when clients are making multiple small requests.

The DVD approach makes the same basic observation as NASD: removing a fileserver from the datapath will improve performance and helps provide a good start in addressing security concerns. While the basic premise is the same as NASD, adopting the object interface in NASD allows the NASD system to avoid some of the message exchange overhead necessary in DVD, as well as to place more functionality beneath the interface.

Chapter 3: Survey of Access Control Policies

One of the goals of the NASD architecture is to provide a mechanism to implement any network storage system. There are a wide variety of different network storage systems in use today and these systems all have different access control policies that determine who can access data. The NASD architecture must allow these diverse network storage systems to implement their access control policies otherwise the NASD system will fail to gain wide acceptance.

This chapter explores a variety of access control systems that I expect developers might build on the NASD architecture and provides some insight into the variety of policies that application filemanagers may want to impose on a network attached storage system. While I include some general information about the various applications' access control systems for background purpose, the emphasis in this section is on describing the parts of the security policy that are discernible to the end users, rather than describing the underlying protocols. This defines the kind of behavior that NASD must support without dictating how the behavior will be implemented.

I describe a variety of different systems that could reasonably be implemented on the NASD architecture. I focus on the behavior of network file systems — specifically CIFS on NT, AppleShare, NFS, AFS, and Novell, because they are widely used and well-known systems. I also discuss the behavior of the OpenVMS implementation of Files-11, the local and cluster-based filesystem for VMS systems, because its complexity offers a stark contrast to the simplicity found in the other systems. Additionally, I discuss the access control in the SQL-92 database standard in order to both explore some of the differences between a filesystem's view and a database's view of access control and examine how this impacts NASD. I will begin with a brief discussion of Multics, which defined many of the desirable security characteristics of a filesystem. As a group, these systems present a reasonable, and hopefully representative, sampling of the systems that may be implemented on the NASD architecture.

3.1 Multics

Multics (Multiplexed Information and Computing Service) [Organick72], which started as a research project at MIT in 1964, is one of the most influential computer systems in history. Being one of the earliest timesharing computer systems, Multics also led to security research and a version called Secure Multics that eventually received a B2 certification from the U.S. Government. The Multics system addresses many of the concerns about data storage that are commonplace today. In [Daley65], Daley and Neuman describe the filesystem in detail and define a set of goals for the filesystem which are still applicable today:

- Safety from someone masquerading as someone else;
- Safety from accidents or maliciousness by someone specifically permitted controlled access;
- Safety from accidents or maliciousness by someone specifically denied access;
- Safety from accidents self-inflicted;
- Total privacy, if needed, with access only by one user or a set of users;
- Safety from hardware or system software failures;
- Security of system safeguards themselves from tampering by non-authorized users;
- Safeguard against overzealous application of other safeguards.

All of these goals are relevant to current filesystems and some have gained greater meaning in the context of distributed systems, where data must be protected from adversaries on the network as well as local users. These are important goals to keep in mind when building any storage system and serve as an inspirational set of guidelines.

Multics uses access control lists (ACLs) to define security policies that control access to filesystem objects (files and directories). A Multics ACL contains a list of users and a set of rights granted to each user. In the more general case, the access control list can also contain a name of a group and a set of rights to be granted to members of the group. In Multics terminology, the set of rights is called the *mode* and it can include up to five attributes: TRAP, READ, EXECUTE, WRITE, and APPEND.

The access control list is processed in order of recency and the first entry applicable to the user in the ACL is used for the request. This is an example of a *short-circuit* approach to access permission processing. Multics explicitly gives more recent entries in the ACL priority over older entries and will automatically delete the older entries when they are no longer accessible.

The most unusual feature of Multics’ filesystem security is the TRAP attribute. For each user listed in the ACL, Multics can associate a list of routines called a traplist. When the TRAP attribute is set to on, if a client attempts to access a file the routines in the traplist will be called in order and each routine is given information about the file or directory being accessed, the identity of the requestor, and the calling sequence which ended in the call of the trap function. When the trap function returns, it reports the effective access mode for the user, which takes precedence over any value explicitly included in the access control list. The TRAP attribute can be used to provide an arbitrary function on a per-request basis such as locking, audit trails, or restricting the accessible portion of a file.

3.2 Network File System (NFS)

NFS is one of the oldest and most widely used network filesystem. Initially designed by Sun Microsystems in 1983, NFS was opened up to the public through an informational internet RFC [Sun89] and was implemented in the early Berkeley Unix systems. NFS provides a *peer-to-peer* file sharing mechanism where a server exports some or all of its storage to the world. Now, almost every Unix-based platform ships with NFS support to import and export storage systems. Sun defined NFSv3, the basis for this description, in 1995 in Internet RFC 1813 [Callaghan95]; future versions are evolving under the auspices of the IETF.

According to RFC 1813, “The NFS version 3 protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server will do normal operating system checking...”. The one exception is that the owner of a file is always permitted to access a file regardless of the permissions. This exception is motivated by the common scenario of a user opening a file and then intentionally changing permissions to prevent other users from accessing the file. Because NFS is a stateless protocol, it is unable to record the fact that the file was open before the permissions changed. This exception is necessary to allow the owner to continue accessing the file as if she had opened it before the permission change occurred.

In practice, most NFS fileservers attempt to provide normal Unix filesystem semantics; NFS was originally developed in Unix environments, so clients have come to expect Unix-like behavior. For Unix-based servers, this behavior allows a server to rely heavily on the access control mechanisms of the local filesystem when servicing an NFS request. Additionally, users see behavior from NFS-mounted filesystems which are very similar to a locally mounted filesystem. Most NFS servers allow additional restrictions to be placed on the exported filesystem, such as exporting it as a read-only file system or remapping the userids of accessing clients. Frequently, NFS servers will also limit the set of hosts that can mount the exported filesystems based on the host name or IP address of requesting clients, although adversaries can easily circumvent this protection.

The NFSv3 specification allows three kinds of authentication mechanisms to be used with NFS: AUTH_KERB, AUTH_DES, and AUTH_UNIX. AUTH_KERB uses a special version of NFS that has been modified to interact with a site’s Kerberos [Neuman94] infrastructure. Most NFS implementations do not support AUTH_KERB, in part, because the security code relies on cryptography that is export controlled. AUTH_DES was defined as part of the originally specified RPC package for

NFS [Sun88a] and was based on Diffie-Hellman operations over GF(2^n), which allows fast calculations but is also easier to break [Schneir96]. A few months after being released, this security risk motivated a revision of the RPC package to version 2 [Sun88b]. However, AUTH_DES was not implemented in early publicly available Unix systems and never achieved wide spread use.

Most NFS systems support only the simplest authentication mechanism, AUTH_UNIX. In AUTH_UNIX, the requestor sends their user ID and group ID along with reach request without any cryptographic protection. Because adversaries can forge user IDs and group IDs, AUTH_UNIX is insecure. To limit access to certain IDs or reconcile IDs from different administrative domains, the server may map the user ID or group ID provided in a request to a different value before checking access rights which limits the set of user IDs that a remote attacker can use when attacking the NFS system.

For each file or directory, NFS associates a user ID, group ID, and permissions. Permissions are defined in terms of mode bits for all the pairwise combinations of {*user*, *group*, *other*} X {*read*, *write*, *execute*}. If the requestor is the object's owner, the requestor is granted the permissions of the user mode bits. If the requestor is not the owner but is currently a member of the object's group, the requestor is granted the permissions of the group modes. Finally, if the requestor is neither the owner nor in the owner's group, the requestor is granted the permissions of *other*. Even if the mode bits for group or other are more permissive than the owner's mode bits, the owner will only be granted the permissions in the owner's mode bits.

NFS uses the normal group mechanism of Unix systems, and the group ID is included in each request. NFS does not do any membership checking at the server since the client machine is assumed to have verified the requestor's group membership. As mentioned in Section 3.2, the fileserver may remap user IDs or group IDs in order to enforce certain security policies or interoperate between different security domains (although this depends on the data provided by the client).

Many sites use NIS [Garfinkle96] or a similar product to maintain a consistent set of user IDs and group IDs across an entire site. With NIS, the client is still providing the user and group IDs to the server; a corrupted client can provide bogus data but well-behaving clients will share a consistent set of IDs. In this respect, NIS is between NTFS (which has a site-wide ID resource) and AppleShare (which does not have very good support for multi-machine synchronization).

NFS has no specific support to provide a set of clients with a common name space. A client chooses where to mount an NFS filesystem in the local tree and what name to associate with the mounted filesystem. Many sites will have a standard naming scheme and mount location to simplify user's lives but this is strictly done by convention (or system administration tools) rather than being intrinsic to NFSv3.

NFSv3 is a stateless protocol so servers maintain no state about active clients and access permissions must be checked on each operation as opposed to at open-time which

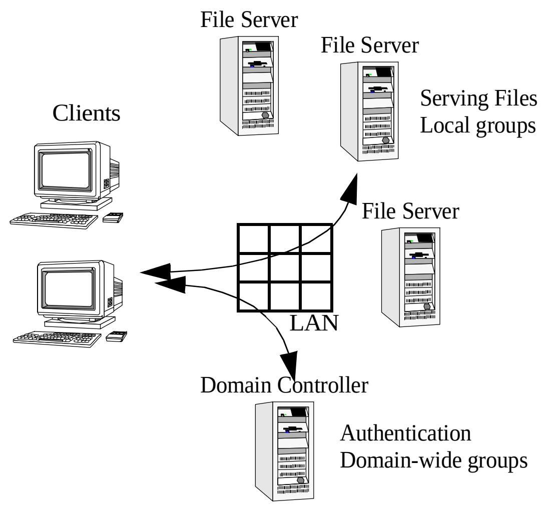


Figure 3-1 NT 4.0 Filesharing Architecture

A user authenticates to a domain controller and obtains an access token that includes their personal security identifier (SID) as well as the security identifiers of any domain wide groups to which she belongs. The user then uses another mechanism to locate the fileserver on which their data is located and uses the access token to open a connection with the server before requesting any data.

is the normal Unix behavior. If a client performs an operation that generates a large number of NFS requests, such as a 4MB file write, then the operation may fail midway to completion if the permissions on the file are changed during the operation.

NFS does not dictate a policy for the inheritance of access control information. The NFS operations to create files and directories both include an attribute field in which a client specifies the initial mode bits for the new filesystem object. Most NFS clients will assign the initial mode bits to an NFS filesystem object exactly as if the object had been created locally.

3.3 Windows File Sharing

Microsoft's Window NT is important because it has high visibility in the marketplace. As long as it is not attached to a network, NT 4.0 has received a C2 security certification under the Department of Defense's orange book criteria [DoD85].

In this section, I paint a picture of Windows filesharing that is built from information gathered from experimentation with NT 4.0, third party NT administration and security books [Rutstein97, Frisch98], the Microsoft Developer's Network Web Pages [Microsoft98], and the CIFS 1.0 specification [Leach97b]. I have access to neither the source code, as I do with NFS and AFS, nor explicit functional specifications, as I do with AppleShare. This is an important limitation of this section because I only present what people outside Microsoft believe NT should be doing and how I have observed NT filesharing to behave. However, it enables me to present a good description of the system and to discuss relevant NASD issues.

When a user first logs into an NT client machine, she authenticates with a domain controller as shown in Figure 3-1, using the NT LAN manager protocol (soon to also support Kerberos [Microsoft96b]). The user receives an access token which contains the

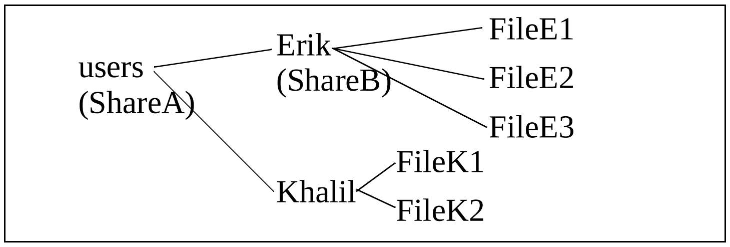


Figure 3-2 Example Directory Structure

user's security identifier (SID) and SIDs for each domain-level group to which the user belongs, as well as any special rights granted by the domain controller. A domain controller is a trusted machine, usually with a mirror backup, that manages the user and group databases for an organization and provides a centralized security and authentication resource for the organization.

While the domain controller provides authoritative information on domain-wide groups, individual fileservers may also maintain a local group database. Any fileserver in the domain may have an ACL that refers to a domain-wide group, while only the local fileserver can reference groups defined on that server. This allows group management to be done at both a local level and at a centralized server.

NT implements ACLs on three filesystem resources: a share, a directory, and a file. The first level of security checks is at the share level. A *share* is a subdirectory of a disk being exported over the network, the network-visible name for the subdirectory, and an access control list protecting the subdirectory. A share can be configured so the name is not advertised on the network. This serves as an extra barrier to a casual vandal but will not deter a sophisticated adversary who discovers the name. When a client connects to a share, the server checks which rights the client has on the share. For every SID in the user's access token, for both users and groups, there may be an entry in the share ACL with one of the following permissions:

- No Access: The requesting entity is denied any form of access.
- Read: The requesting entity can potentially read or execute files stored in the share.
- Change: The requesting entity can potentially read, write, execute, and delete both files and directories in the share
- Full Control: The requesting entity can potentially have all the rights as in Change plus the ability to modify access permissions or take ownership of files and directories.

Share permissions act as a ceiling on what operations the client will be able to perform on all files and directories accessed though the share. Consider the example in Figure 3-2: if a user only has read access on *ShareA*, the exported name for */user*, then the user will never be able to modify or create files in */user/khalil* even if the ACL on */user/khalil* grants the user unlimited access rights. However, it is possible to export two directories, one a subdirectory of the other, and have different rights depending on the path

ACL for directory */user/Khalil/FileK1*

PDLCoWorkersRead
PDLUnderGradsNone
TrustedFriendsWrite

Figure 3-3 Example NT ACL

through the file system. In the example, if a user has only read privileges on *ShareA* but full privileges on *ShareB* then the user will never be able to create files in */user/Erik* if she accesses the directory via *ShareA*. If the user access */user/Erik* through *ShareB* then she potentially has the ability to create files in */user/Erik*. This illustrates that a filesystem may provide security at a variety of levels of the system rather than simply on directories or files and that the interaction of these layers must be properly captured in an implementation on the NASD architecture.

The next level of protection in NT, which is also applied to local users, are directory or file permissions. Directory permissions restrict a user's ability to gather information about the contents of directories including the file identifiers and the filesystem's internal name for its files. Protecting a directory will make it harder for an adversary to find a file but will not prevent an adversary from accessing the file if she knows the file identifier. However, for normal usage with unmodified clients, the directory permissions help provide security to the files contained within the directory. In NT, as well as NFS and AFS, the directory permissions hide the underlying filesystem name for a file from an adversary but do not prevent an adversary from accessing a file if she knows the proper name to use.

In NT, this default behavior is the result of clients being granted a special right in their access token called **Bypass Traverse Checking**. This special right allows the users to perform an operation on a file or directory based only on the permissions of the file or directory they are accessing and the share permissions. Without this right, NT will traverse the path from the root of the share to the accessed object and verify that the client has permissions at every step along the way. If nobody has the **Bypass Traverse Checking** right then an administrator may change an ACL on a directory and have it immediately protect the directory's descendents. If users have this right then, as long as the know the name of the file or directory, they can access the file or directory regardless of the ACLs in the ancestor directories.

All ACL systems need rules to determine a user's rights when multiple ACL entries are applicable. Consider the ACL for */user/Khalil/FileK1* given in Figure 3-3. If a user is both a member of *PDLCoWorkers* and *PDLUnderGrads*, then which rights should the user be granted? In NT, she will be denied any access to the file. For that matter, all users who are in *PDLUnderGrads* will be denied access regardless of any other group memberships. If a user is a member of *PDLCoWorkers* and *TrustedFriends* then she will be granted both Read and Write access to the file. These decisions are captured by the following two rules:

1. If a user or group is explicitly denied access, i.e. an ACE set to No Access, then no access is permitted. Negative rights have precedence over positive rights.
2. If a user is not explicitly denied, she is permitted the sum of all the rights she is granted by access control list entries.

While the underlying idea of an NT ACL is similar to Multics, NT does not halt evaluation when the first applicable ACL entry is found. Instead, NT prioritizes explicit denial of rights over explicit granting of rights but otherwise grants a client the sum of all their rights in the ACL.

NT filesharing uses a *peer-to-peer* sharing model, which is similar to NFS, where individual fileservers choose to export a portion of their local filesystem to remote clients. A group of shares exported by different NT machines within a domain have no structure other than ones created by shortcuts, essentially symbolic links, in the local filesystem and well known locations. Any structure built out of symbolic links is tightly tied to the underlying names of the servers and it is difficult to move parts of the filesystem to different servers without involving the clients in the move.

With NT 5.0, Microsoft plans to include the Microsoft Distributed Filesystem (Dfs) which adds a layer of glue-logic over the peer-to-peer systems to provide a shared logical namespace [Microsoft96a]. Dfs does not add any new security services but allows improved management of storage in an enterprise setting. The Dfs design caches mappings of path prefixes to servers so clients do not need to repeatedly traverse the namespace. However, this implies that a client who lacks the **Bypass Traverse Checking** will still bypass some permissions checking on a requested object's path when using cached prefixes. Permissions will only be checked on servers the client must traverse because their prefix to server mappings hasn't been cached and at the final server holding the requested file or directory. Even though NT supports strong security by verifying permissions along a path, Microsoft is restricting the utility in their design of Dfs.

Unlike many file systems, NT 4.0 has built-in support for audit logs. Any user, typically security administrators, who has the **Security Privilege** right is able to view and clear audit logs as well as set the System ACL (SACL) on any NT object, including files and directories. The SACL specifies SIDs and the operations to be logged. NT is able to log the success and/or failure of read, write, execute, delete, change permissions, and take ownership operations. The SACL is inherited through the normal filesystem inheritance system but *cannot* be modified by someone without the **Security Privilege**, even if that person is the file's owner. Since NT is using a peer-to-peer namespace, there is no overreaching orchestration to synchronize audit logs of requests made to multiple fileservers. Tools built on top of the filesystem logs must provide this synchronization.

Windows' file sharing is based on the CIFS 1.0 specification which specifies reliable connection-based transports. The CIFS protocol specifies a stateful protocol in which the server is notified when a client connects to, i.e. mounts, a share, along with the opening and closing of files. NT fileservers appear, from experimentation, to check permissions at the time a file is opened. Presumably, this is a performance optimization to avoid repeatedly checking the permissions on every request. Another result is that the effect of ACL changes will become evident to clients only when the client next opens the file rather than on the next I/O operation.

NT supports groups that are either local to a server or global within a domain and these types of groups have different behavior when group membership changes. If a user is given an access token that says she is in the domain group *Students412* then she will be able to use this token until it expires, even after she is removed from the membership list at the domain controller.

In contrast, local group membership changes take effect whenever the client next connects to the fileserver. The CIFS specification recommends that connections be closed after they have been idle for a minute although this is invisible to the user. When a client machine reconnects to the server, the user will see the effect of any ACL change or modification to the local group membership database but will not see the effect of a change in domain group memberships unless she has also re-authenticated to the domain controller.

The CIFS 1.0 does not specify when access permission checking should be done but does allow read and write operations to return **ErrNoAccess**. Strictly speaking, a CIFS-compliant server, although not emulating NT, could verify permissions on each operation rather than at open time as NT 4.0 appears to do. If the CIFS-compliant server verified permission per operation, the server would be a valid CIFS server but clients may not interoperate well with a server which behaves differently from the standard Microsoft servers.

When a file or directory is created in NT, it inherits the ACL from its parent directory. This is called *static inheritance.* Once an ACL is inherited from the parent, the parent’s ACL can change without affecting the ACL of the child. This is the easiest type of inheritance to implement because the server only consults the requested directory or file ACL rather than evaluating a dependency on its parent directory. This is both easier to implement and more efficient for accesses since the server processes one, as opposed to multiple ACLs.

3.4 AppleShare

AppleShare [Poole97], a standard feature of all versions of the Macintosh OS, enables *peer-to-peer* file sharing. This section of the dissertation describes the mechanisms used in both Mac OS 8 and AppleShare IP 6.0. These products share the same model of security and file sharing; additionally, AppleShare IP includes WWW and FTP servers, both which could be supported by NASD devices, along with a higher-performance fileserver. Both Mac OS 8 and AppleShare IP 6.0 implement the Apple Filing Protocol (AFP) version 2.2. [Sidhu90, Apple98].

Appleshare has three categories of access privileges for network users: *owner*, *group or user*, and *everyone* on a per folder (Macspeak for ‘directory’) basis. Each directory has an *owner*, either a specific user or a group, who is the only user or set of users that can change the permissions on the directory. The *group or user* category enables a single group of users to be granted access rights on the directory. If a user is granted rights under both the *owner* and *group or user* categories then she has the union of permissions associated with both categories of users. The *everyone* class is a catch-all for all other users allowed to connect to the file server. The Mac user interface will not allow administrators or owners to set any of the user categories’ rights lower than the rights of *Everyone* because *everyone* rights logically serve as a floor on the rights granted to any user.

In a Mac environment, a server views users as either *registered*, having a user ID and possibly a password, or *unregistered*. To prevent unknown users from accessing data, a Macintosh can be configured to not allow operations by unregistered users. If unregistered access is disallowed, the rights granted to *everyone* apply only to registered users who are not otherwise explicitly granted rights.

The AppleShare Filing Protocol specification [Sidhu90, Apple98] assumes a connection-based protocol and specifies that permissions are checked at open time. The actual data moving operations do not check permissions but rather rely, as users expect in most local filesystem, on the rights granted at open time.

The statefulness of the AppleShare Filing Protocol enables an AppleShare server to allow in-progress operations to complete even when an access control change would prevent the operation from beginning. For example, if a user is copying a file from a AppleShare server to the users’s local machine and the server owner decides to revoke the users’s access rights, the copy will be allowed to complete unless the entire server is shut down.

AppleShare servers define groups locally to the server machine since AppleShare has no notion of an administrative domain. In order to allow coordinate multiple AppleShare servers, some versions of AppleShare provide support to easily export the entire groups or passwords database from one server into a file and to incorporate the database into another. However, there is no centralized or on-line service providing authoritative service for group memberships as is found in Windows NT, AFS, or NIS, which is frequently used in NFS sites.

AppleShare does not provide the global shared namespace found in systems such as Microsoft’s Dfs and AFS. It is designed to provide basic sharing semantics that allow a user to export, with some measure of security, a portion of a disk to be accessed by remote users. In this manner, it is more similar to NFS than a modern distributed filesystem.

Unlike most of the filesystems surveyed, AppleShare servers support *dynamic inheritance,* which requires that the inheritance of access protection occur *at call time.* A directory can be configured to inherit permissions from its parent directory which requires that the directory tree be walked upwards at request time until an ACL, rather than a reference upwards, is found. Walking the directory tree on each request is, potentially, an expensive performance issue because of the amount of data that must be consulted. The advantage of dynamic inheritance is that it allows rapid changes of permissions on a large group of objects. If the permissions on the top of a directory tree are changed, all future accesses to lower level filesystem objects are affected.

3.5 Andrew File System

The Andrew File System (AFS) was designed at Carnegie Mellon University in the 1980s as a solution to scalability and performance problems of existing network file systems [Howard88, Satyanarayanan89] and was transitioned into industry by Transarc Corporation. This section describes AFS based on a combination of reading early AFS papers [Howard88, Satyanarayanan89], examining AFS documentation [Transarc92], inspecting source tree for Transarc's AFS Version 3.4.p2, and experimenting with CMU's departmental AFS servers.

AFS implements a *common-namespac*e model where a client machine interacts with a set of servers, called an AFS cell, that collectively provide the AFS service to clients. Together, the servers provide the illusion of a larger common filesystem shared between clients where the name of the server on which data is located is hidden from the user and can be changed without affecting a user's ability to access their data. In addition, multiple AFS cells from across the world can be mounted with AFS to create a wide area shared file system.

One of the strengths of AFS is that it to addresses the security shortcomings of its predecessors like NFS. A good description of AFS's security system can be found in [Satyanarayanan89]. Early versions of AFS used the Andrew Secure RPC, which Burrows et al. raised concerns about [Burrows90], but the core security design has remained unchanged. An AFS client first authenticates via Kerberos before contacting the AFS servers and obtaining *tokens*, also called AFS tickets, that are used for future communication with fileservers. The user must implicitly trust the client workstation not to misuse its tokens, but the AFS cell places no trust in the client workstation.

In AFS, access control information is stored in per directory ACLs. Each ACL lists up to 20 access control entries (ACE) which contain a user ID or group ID and a set of rights. There are two classes of access control entries: positive or negative. A positive entry explicitly grants rights to a client while a negative entry explicitly prohibits a client from obtaining specific rights. Negative entries always take precedence over positive entries so it is easy to prohibit a user, or group of users, from accessing some data.

```
Access rights for .
Normal rights:
    hgobioff rlidwa
    hgobioff:users lia
    hgobioff:friends rl

Negative rights:
    hgobioff:enemies rlidwka
```

Figure 3-4 Example AFS ACL

AFS has the following access rights:

- **Write** - The user can modify the contents and mode bits of files in the directory.
- **Lookup** - The user can list the names of files and subdirectories contained within the directory and view the directory's access control list.
- **Delete** - The user is allowed to delete files from the directory.
- **Insert** - The user is allowed to create new files in the directory.
- **Lock** - The user can issue the *flock()* to get advisory locks on files in the directory.
- **Administer** - The user has the right to modify the directory's access control list.

When checking a user's permissions, AFS examines all the access control entries within an access control list and grants users the union of their normal rights minus the union of their negative rights. The result is that negative rights take priority over positive rights. Consider the example is Figure 3-4, if a user is a member of *hgobioff:friends* then she will only be able to lookup and read the files in the directory and if she is a member of *hgobioff:users* she will be able to lookup, add new files to the directory, and append to files. If the user is a member of both groups then she will be able to lookup, read, insert, and append. However, in all cases, if a user is a member of *hgobioff:enemies* then she will not be granted any rights because all rights were prohibited by the negative ACL entry. AFS is very similar to NT with respect to how ACLs are handled because both give negative ACL entries precedence over positive entries and grant the user the sum of their rights rather than using short circuit evaluation as Multics does.

The rights granted by the ACL are only the *potential* rights of users files. AFS uses the Unix mode bits to further refine the rights given to clients. Read operations require that the read or execute mode bit be set and write operations require that the write mode bit be set. Normally, the owner mode bits are used when the file's owner attempts an access while the group mode bits are used for all other users. However, AFS servers can be compiled to use the owner mode bits to check access for all users.

Each AFS cell includes a protection server that maintains a list of groups defined in the cell. Users create groups with names of the form *UserName:GroupName* by issuing appropriate calls to the protection server. Privileged system administrators can create groups with the form *GroupName.* Once a group is created, any user can reference the group in an access control list to grant or prohibit rights to the group's members. Although anyone can reference the group, the ability to view group membership is controlled by the permissions set on the group. Because the group database is cell-wide, all users can take advantage of well-known groups.

AFS groups can contain either user IDs or IP addresses that have been specifically defined to AFS. IP address authentication is a very weak form of authentication because an attacker can easily spoof an IP address. In contrast, a user who is a member of a group through their user ID has been strongly authenticated to AFS using Kerberos before she is allowed to exercise her rights.

When a user first authenticates to AFS, she is given *tokens* which encode her user ID and groups ID. AFS servers check a client's group membership by examining the tokens, rather than consulting with the cell group database server. Because the tokens are given to a client when she first authenticate to a fileserver, any changes to the groups that a user belongs to will not take effect until the user next authenticates to the fileserver. In an average AFS site where tokens are valid for 24 hours, there will be a latency of up to 24 hours before a group change will take effect, unless the user voluntarily re-authenticates to the server.

AFS performs a local access check to decide to allow an operation. The client's AFS token is compared to the ACL of the directory being accessed, which is the parent directory for a file operation, and against the mode bits of the file or directory being accessed. AFS has no provision to force clients to have valid permissions along an entire path, so a client holding the unique file identifier for a directory can access it even if a higher level directory (or in the case of a file, a directory other than its parent) disallows accesses. This means that a client can't walk through the filesystem namespace to find out the file identifier of a file, but an adversary who observes this on the network can utilize it to access files below if a user relies on directory permissions at upper levels of the directory hierarchy to protect the lower levels.

AFS uses *static inheritance* of the access control lists on directories and a very simple form of *dynamic inheritance* for files. When a directory is created in AFS, the default behavior is for the new directory to inherit the ACL that the parent directory has at the time of creation. Future changes to the parent directory will only impact a client's ability to traverse the name space but not the ability to directly access the new directory. Because files do not have ACL entries of their own, presumably as a space saving and simplicity measure, the permissions of a file are dynamically inherited from the parent directory at request time. If a parent directory's ACL changes then the change immediately affects the directory's child files. The child directories may be more difficult to find if lookups are prohibited, but actual operations on child directories will not be affected by the change to the parent directory's ACL.

3.6 Novell Netware

Novell Netware 4.1 includes both the Novell Directory Service (NDS) and file servers [Sheldon96, Steen96]. In contrast to the other systems I've surveyed in this section, Netware is tightly tied to an enterprise wide directory service which provides several abstractions: containers — objects that contain other objects, and leafs — groups, users, and printers, in addition to filesystems.

The directory is a scalable hierarchical namespace/database that users consult to find critical information about enterprise resources such as fileserver network addresses, group membership lists, and user phone numbers. This is similar to the AFS namespace but is object oriented rather than file oriented and it is used to store typed data such as user descriptions. In order to find a resource, a user must traverse the directory to find out where it is located in the network. This provides a very basic level of security in hiding the existence and names of resources through the object protection in the directory. However, the protection in the directory is primarily focused on protecting data stored in the directory while data stored in the filesystem is protected through local filesystem protections which can depend on data stored in the directory.

A user is primarily granted access rights by their position in the directory (remember, users are objects) and their group memberships. A user is granted permissions that were either directly given to her or granted to one of her parent containers. For example, if a user is identified by the location **USA.SF.Devel.Amiri** in the directory then this user would be given rights explicitly granted to him as well as rights given to **USA**, **USA.SF**, and **USA.SF.Devel**. A user's location within the directory will normally follow an organizational structure of a company although this is not a requirement.

A user may also have rights associated as a result of being listed as a member of a group which is an explicit list recorded in the directory and is distinct from a user's position in the directory. Groups are not bound by the hierarchical structure of the directory service and can be used to associate an arbitrary responsibility or role with a user similar to the group mechanism in AFS.

Novell uses ACLs on both files and directories. An ACL entry lists a *trustee*, which is a user, container, or group and the rights granted to the trustee. Unlike filesystems such as NT file sharing where file rights take precedence, Novell file rights are strictly additive to directory rights. If a user is granted a right at the directory level then she will receive this right on all the files in the directory regardless of file ACLs. If a user is granted a right on a specific file and not on the directory then she will still receive the right for the file.

Novell files and directories inherit rights granted to a trustee in any of their parents. Unless specifically prohibited, which I will discuss next, a client can access a file if she, or one of her ancestor containers or groups, is granted read access to any of the directories along the path to the root. This is a very far-reaching form of *dynamic* inheritance because the entire path may need to be examined to determine a clients access rights.

Novell rights can be curtailed by either explicitly setting rights for a trustee or using an Inherited Rights Filter (IRF) on a directory. If a trustee is explicitly assigned rights on a directory or file, the explicit rights will be used at that level rather than inherited rights — with the exception that file rights are always additive to directory rights. The IRF is a bitmap that says which permissions can be inherited by this level and its descendents. If the IRF for a right, such as read, is not set on a directory then no trustee can inherit that right through the directory. For example, if the IRF on **/usr/apples/jim** does not allow for read permissions to be inherited then a user will have access to files in **/usr/apples/jim/build** only if she is explicitly granted permission on **/usr/apples/jim**, **/usr/apples/jim/build**, or the actual file. Any read permissions that the user was granted on **/usr** or **/usr/apples** will be ignored. The only exception is that a user granted supervisor rights on the volume (e.g., the root of the filesystem) can not be blocked by the IRF.

3.7 Files-11 and OpenVMS

In this section of the dissertation, I present an overview of the access control system used in Files-11, which runs on OpenVMS AXP Version 1.3 and OpenVMS VAX Version 6.0 [DEC93a, DEC93b]. This access control system has evolved over the last few decades into a significantly more complex and rich access control scheme than the other systems that I have described.

Files-11 is used both as a stand alone filesystem and in tightly knit clusters of workstations. The information sources I have available are primarily user and administrator documentation [DEC93a, DEC93b] rather than a specification of the underlying protocol or source code; this limits the amount of detail I can provide. My goal is to present the reader with some feeling for the complexity of this well-established commercial access control system, not a complete specification of the behavior of OpenVMS and its filesystems. Because of this limitation, I will focus on presenting the view of filesystem access control that OpenVMS and Files-11 present to local users.

3.7.1 A User’s Security Profile

When a users is logged into an OpenVMS system, their processes have security profiles that contains a *user identification code (*UIC*)*, *rights identifiers*, and a set of *privileges* that collectively define the operations that the user can perform within the system. A UIC defines a group membership and user name which is unique for each user on the system (e.g. [USER, MOLLY]).

The *rights identifiers* define attributes of the user’s current process. Broadly, there are three major classes of rights identifiers: environmental identifiers, general identifiers, and UIC identifiers. Environmental identifiers are defined by OpenVMS and describe characteristics of the process such as BATCH (a batch processing job), NETWORK (a user connected over the network), DIALUP (a user connected through a dialup pool), etc.

General identifiers are defined by the site security administrator and describe a characteristic of the user, such as HELP_DESK_STAFF, SVC_ENGINEER, SALES_REP, or PR_STAFF, who owns the process. The general identifiers have a role similar to groups in many distributed filesystems but differ in that they are a characteristic of the user rather than a membership list of the group. Finally, UIC identifiers are restatements of the user's group affiliation and name. If a user is entitled to a specific set of rights, the user can choose to enable or disable these rights, which enables a user to operate in different roles based on their currently active rights.

Privileges are a special set of identifiers defined by the system that allow access that would be otherwise prohibited to a user. They play a similar role as the special rights, such as **Bypass Traverse Checking** and **Security Privilege**, in NT. They allow bypassing or overriding the normal security protection in ways that the OpenVMS designers have found useful to users and administrators. As with rights, a privilege can be enabled or disabled as needed by the user, which allows this powerful feature to be used only when absolutely necessary. An important difference between privileges and rights is that privilege behavior is hard coded into the operating system with predefined effects, while rights are determined by how administrators or users configure the protection in the filesystem.

3.7.2 Protecting Files and Directories

The security of a file or directory is defined by its security profile which contains an owner, a protection code, and an optional access control list. When a user performs an operation, the users's security profile is evaluated against the security profile of the file or directory to determine if the operation should be allowed.

Together, the owner and protection code define a very basic security mechanism much like the user ID, group ID, and mode bits in Unix filesystems. Protection codes define the access rights of four types of users: the *owner*, users who share the same *group* UIC as the owner, all users on the system — i.e. the *world* — and users with *system* privileges or rights. *System* users are users with a userid less than some site-specific parameter, the owner of the volume on which a file or directory resides, users with the SYSPRV privilege, or users with the GRPPRV privilege who share UIC groups with the objects owner.

OpenVMS permits an operation if the requestor qualifies as any of the four types of users. This means that *world* access rights serve as a floor on the rights granted to any user. If the *group* and *owner* rights are set lower than world, users who are members of the owner's UIC group or are the owner will still receive the rights given to *world*. This is different from most Unix systems where short-circuit evaluation is used and the *owner* or *group* user would be denied access.

Access control lists enumerate specific combinations of rights identifiers and the access rights granted to processes with the rights identifier. Because an ACE can refer to a combination of rights identifiers, users can specify not only a specific set of clients that are granted rights but also some environmental characteristics of their access. For example, a user can specify that only another user named MOLLY can access a file only when connected over the NETWORK by making an ACL entry for MOLLY+NETWORK.

Unlike AFS where a user is granted the sum of their rights in an ACL with negative rights taking priority, OpenVMS relies on the ordering of an ACL to control access. The ACL is processed sequentially and a user receives the rights of the first applicable access control entry. Administrators or users can restrict the rights of another user or group of users by adding a more specific and restrictive ACE early in the ACL. A user is explicitly denied access if their identity matches an ACE which specifies **NONE** as the access rights.

In addition to the basic access rights, an access control list can also grant control access to a filesystem object which allows the user to change the protection code and the ACL. Additionally, users have control access if they have the same UIC as the owner or qualify as system user.

The last mechanism for a user to gain access to a filesystem object is through specific OpenVMS defined privileges. The following are the privileges that can override security protection on an object:

- BYPASS - A user has full access to all filesystem objects.
- GRPPRV - A user with this privilege whose UIC group matches the owner of the object receives the same access given to *system* users. This includes control access so the user can freely manipulate any object owned by a user in the group. This allows a user to have full access to objects created by users in a certain administrative group, defined by the UIC group identifiers.
- READALL - The user is granted read access to all filesystem objects.
- SYSPRV - The user is granted access given to *system* users. This includes control access so the user can freely manipulate all objects in the system.

Access rights are verified in the following order:

1. Check for a matching ACL entry
2. Check the protection code
3. Check for special privileges

If access is not granted at one step, verification of access rights continues to the next step. If the protection code is more permissive than the ACL, the a user will always be granted access through the protection code; meaning that the ACL will not be effective. The special privileges are defined by OpenVMS and the user can not modify the rules that determine when they apply. In general, a given user probably wants to use either protection codes or ACLs to control access to filesystem objects rather than a delicate mix of the two mechanisms.

In addition to the protection mechanisms for files and directories, administrators can also set similar protection on a volume basis. In order to access a file or directory, a user must have appropriate privileges on both it and the volume in which it resides. This is very similar to Window's share level protection.

OpenVMS includes extensive support for auditing either on a system wide basis or on a per-object basis. An administrator can configure the system to generate an audit log of all events of a specific class, such as using GRPPRV privileges to access a file or reading a file, or may include an ACL entry on an object that generates audit records when the object is accessed. In addition to normal audit records, OpenVMS also can log an event directly to the operator's terminal, which is called an alarm. These features allow administrators to track the use of critical resources and the exercising of specific privileges within the system.

3.7.3 Inheritance

A security profile for a new object can come from several sources which are, in order of precedence: an explicitly specified security profile at creation time, the profile of the previous version of the file, a profile inherited from the parent directory, or a default protection code from the process creating the file or directory. The default protection code for a process is similar to the umask feature in a Unix system.

Like many file systems, Files-11 files and directories can inherit permissions at creation time from their parent directory. Unlike many file systems, these permissions are explicitly specified as part of the parent directory's security profile instead of being implicitly defined as being equal to the parent's permissions. To specify that an ACL entry should be inherited by its children, a user creates an ACL entry with the **DEFAULT** option set. Similarly, a user creates a **DEFAULT_PROTECTION** entry in the ACL to specify the protection code of any new children. An ACL entry that specifies the permissions to be inherited will have no effect on normal access control decisions. This allows a richer behavior than in AFS, where permissions are directly inherited from the parent, because in OpenVMS the protection applied to children can automatically be made different from the protection applied to the parent.

3.8 Generic Authorization and Access Control API

The Generic Authorization and Access Control API (GAA API) extends the traditional access control list to provide a flexible distributed authorization that allows a wide variety of access control policies [Gheorghiu98, Ryutov98]. The focus of the API is to collect a variety of different policies and express them through a single API.

While a traditional ACL associates rights with a principal, the GAA API extends this to also include a set of conditions that the principal must meet in order to exercise the rights (called an EACL). Some of these conditions, such as time of day and authentication mechanism, are generic and interpreted by the GAA API, while others, such as CPU load or memory usage, are interpreted by the application. These conditions can provide the same type of information as OpenVMS's environmental identifiers by describing qualities of the client other than the client's identity.

Individual client requests are evaluated against the requested object EACL and the security context of the client. The client's security context includes information such as the client's identity, connection state, group membership, and group non-membership. This allows the underlying services beneath the GAA API to make a decision about an access attempt or, if insufficient information is available, defer the decision to the application level. Some of the conditions applied to EACL entries may be application-specific, such as a load metric. Thus, only the application can decide if the condition is or is not met.

The GAA API provides a unifying abstraction for administering security across a heterogeneous set of resources in policies. For example, the creators suggest that the API could be used to control access to servers, remote printers, or large scale multicast applications. By unifying the abstractions, it becomes possible to use the same mechanism across multiple administrative domains and a variety of applications.

3.9 SQL-92

The SQL-92 specification was released jointly by the American National Standards Institute and the International Organization for Standardization as a global standard for database query languages. SQL-92 has two concepts that are critical for understanding its security model: views and grant [Ramakrishnan98].

A view is a mechanism for abstracting away the structure of the underlying data into a standard presentation, but it can also control who can access information by granting a user access to a view of part, but not all, of a table, a data set, or group of tables. A view provides the appearance of a table that does not actually exist. The rows are not explicitly stored in a database but rather are computed in response to a request made through the view. Computing a view can require eliminating data from an existing table or combining information from a multiple tables.

Initially, the creator has all rights on a table or view and nobody else has rights. Rights are explicitly propagated to other users using the grant command rather than inherited through the directory hierarchy which is the filesystem approach. The syntax of the grant command is:

GRANT rights **ON** object **TO** authorization-id [**WITH GRANT OPTION]**

The rights include the ability to perform basic operations on database objects, either a table or a view, such as: alter, delete, index, insert, references, select, and update. An authorization-id is an identifier that may represent either a user or a group of users. If a user is granted a right with the grant operation, then she is able to grant the right to another user.

These characteristics are sufficient for discussion about the basic relationship between a DBMS and NASD security. DBMS systems, unlike filesystems, allow operations to be *data dependent*. A filesystem simply stores and retrieves data while a DBMS provides storage, presentation, query, and conditional operations that can all be hidden behind a view over which clients can be granted rights. The DBMS system has specific knowledge of the structure of stored data, which enables the DBMS to provide the functionality that differentiates it from a filesystem.

A DBMS's awareness of the underlying structure of the stored data directly impacts the nature of security through views. When a user is granted access through a view, the user can be given access to parts of tables which she is otherwise prevented from accessing. For example, a DBMS storing an employee database may grant a manager access to records of people in their department but not those in other departments. Because the DBMS understands the structure of the employee database, the DBMS can provide this *content-specific* access control.

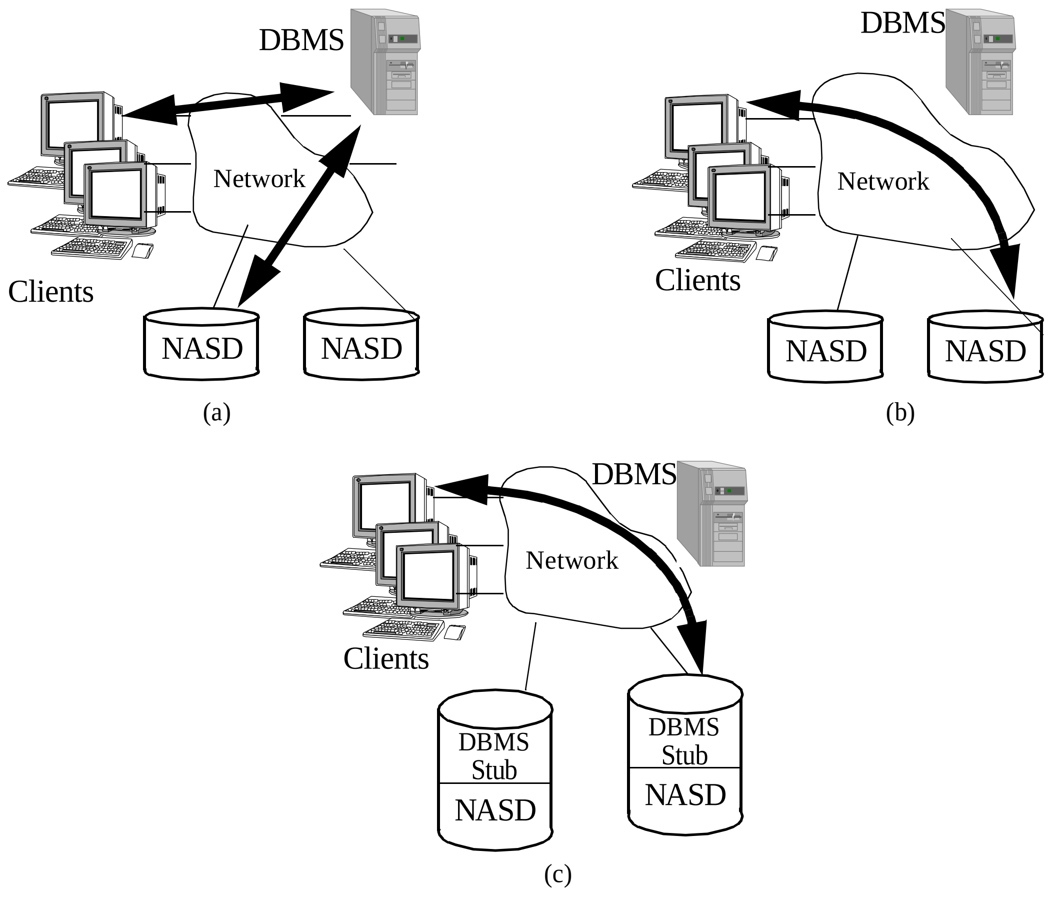
A NASD storage system is unaware of the underlying structure of stored data and thus can not provide this type of security behavior to clients. NASD can provide object-granularity access control and byte-range access control but not the content-specific access control available in most DBMS systems.

In a database implemented on the NASD interface with clients directly accessing storage, the DBMS system can only grant or disallow clients on a per NASD object (i.e. database table) granularity as shown in Figure 3-5b. However, in a three-tier system, shown in Figure 3-5a, a group of database servers can share NASD storage while the servers handle requests from clients. This enables the DBMS servers to gain the scalable aggregate bandwidth of network attached storage, which is shared across DBMS servers, and still provide the *content-specific* access control inherent in database management systems. This architecture relies on the DBMS server CPU and network interfaces to deliver the performance of the storage system to end users. Riedel et al. [Riedel98a] have shown that moving database functionality closer to the storage potentially increases DBMS performance and that this same technique could be used to provide *content-specific* access control at the storage level as is depicted in Figure 3-5c.

The basic NASD access credentials, which are described in Section 4.2.2, can grant access on a byte-range basis which allows a DBMS system a limited amount of flexibility in granting access to clients in the basic NASD system. For example, a DBMS system could store a table in a NASD object with each column behind a byte-range in the object. More concretely, for all rows, column 1 is stored in the first megabyte of the object, column 2 in the second megabyte of the object, and so on. A DBMS system can grant a client permission to directly access certain columns of the database by using byte-ranges. This is a far cry from the *content-specific* access control that a DBMS normally provides,

Figure 3-5 NASD Based DBMS Architectures

Figure 3-5a shows a client workstation accessing a DBMS system through a database server while a group of database servers share access to a large set of NASD drives. This allows the servers to enforce security on every access storage but limits the performance advantages of the NASD architecture. In Figure 3-5b, the client workstations directly interact with NASD storage and infrequently consult a database manager. However, the database server can only grant access to contiguous ranges of a NASD object so this approach limits the freedom of the DBMS security policy. Figure 3-5c shows a client in NASD and ActiveDisk [Riedel98a] system where a portion of the DBMS resides on the disk which could provide some of the DBMS functionality including enforcing security.



but it demonstrates that the features of the NASD interface and a properly structured database system can provide finer access control than at a whole table granularity.

3.10 Discussion

In the preceding chapter, I described a variety of different access control systems that designers may build on NASD. From this survey, I've found several issues that need to be considered when designing NASD's security system and implementing systems on top of NASD: arbitrary security critical code on the control path, rich set of relationships described in access control structures, dynamic inheritance, and promptness of group membership changes when permissions are checked in the application, Table 3-1 summarizes the issues for the distributed filesystems.

- **Arbitrary security-critical code on the control path**: Multics' TRAP function introduces arbitrary security critical code into the control path for I/O operations. Similarly, a DBMS inserts content-specific processing which performs filtering on data presented to the user but is otherwise arbitrary. NASD removes the server, which can implement arbitrary functionality, from the control path, thereby removing the mechanism to execute the arbitrary code in response to each request.

  The idea of adding a general purpose execution engine on the path between the requestor and a filesystem for security purposes may have originated with Multics but other researchers have revisited the idea. Rabin and Tygar explored a similar mechanism in the ITOSS system and demonstrated that it can be used to provide a finer granularity of control than file level permissions [Rabin89]. Bershad and Pinkerton investigated a similar idea called Watchdogs where client requests are intercepted and sent to a user-level filesystem extension program that extends the semantics of the filesystem [Bershad88].

  Because of NASD's relatively simple, (albeit high-level) interface, the only way to provide TRAP-like functionality is involve the file manager on every request. This degenerates into more of a Net-SCSI [Gibson97a] model in which control goes through the file manager and reduces overall scalability by increasing file manager load although data transfers do bypass the filemanager.

  Ongoing Active Disk [Ridel98a] research is aimed at adding to each storage device the ability to execute arbitrary code on a per I/O operation basis. While the goal of Active Disk research is to deliver increased performance to clients, the same technology can be used to extend the storage system semantics to provide the kind of behavior required by Multics' TRAP function, a DBMS, Watchdogs, or ITOSS.
- **Rich set of relationships**: Both OpenVMS and GAA API provide a rich set of attributes that can be used in access control systems and this fact makes it difficult to efficiently implement these systems on top of simple abstractions such as groups of objects. In Open VMS, a client's access rights depend on a complicated interaction of the user's identity and the attributes of their current process, as well as the protection code and ACL for the requested object. The complicated interactions, illustrated by a multi-page flow-chart in the OpenVMS documentation, depend on exactly which privileges are granted to a user. However, simple structures such as groups of objects could handle the common case modes of getting access through an ACL or protection code while the more exotic mechanisms are handled on a case-by-case basis.

Table 3-1 Summary of Survey of Network Filesystems

Filesystem	Group Support	Timeliness of Group Membership Changes	Time of Access Check	Inheritance
NT	Shared/Local	Next Login (Domain) Next Connection (Local)	On Open†	Static
AppleShare	Local	On Open	On Open	Dynamic
NFS	Local*	Next Request (if a local group or NIS is used)	On each request	Static
AFS	Shared	On Login	On each request	Static‡
Novell	Shared	?	?	Dynamic w/filters

† The CIFS 1.0 specification allows the server to return NoAccess in response to a read or write request so a CIFS compliant implementation could check access rights per operation and reflect ACL changes immediately although experiments with NT 4.0 server indicate that it tests at open time.

* An extra layer of sychronization may be used to maintain consistent group membership databases across multiple servers but it is not integral to the system.

‡ Since AFS files do not have ACLs, they dynamically inherit the permissions of their parent directory which behaves just as if they all had the same permissions as the directory ACL.

The GAA API presents a powerful interface for expressing any policy that can be distilled down to an ACL and a set of conditions on the ACL. This provides a feature-rich method of expressing policies that presents a similar challenge to the ones posed by the OpenVMS security structure. In contrast to OpenVMS, the GAA API is specifically intended to operate in a distributed, or metacomputing, environment. From the NASD perspective, the GAA API offers an interface in which complicated policies can be expressed by an application that may exceed the capacity of a storage device.

- **Dynamic inheritance**: In Appleshare, Novell, and, to a much lesser degree, AFS, filesystems use dynamic inheritance to allow small changes in the access control information to have an impact on a large number of filesystem objects by introducing a dynamically evaluated dependency between an access control decision for a requested filesystem object and the access control information of another filesystem object. In NT, when a user does not have the **Bypass Traverse checking** right, NT implements a similar dependency check by verifying a client’s permissions on all directories along the path to a requested filesystem object. Some of the access control mechanism for NASD that I will present in Chapter 5 allow or reject requests strictly based on the contents of the requested object’s metadata and the requesting client’s access credentials, i.e. a local decision. A local decision simplifies the interface to the access control system and strictly limits the amount of I/O necessary to verify a request. With some of these local decision solutions, if access control on a directory becomes more restrictive, a dynamic inheritance policy in a file manager will force the file manager to update the metadata of all affected objects for which access credentials are outstanding.

- **Group membership changes**: When a user is removed from a group, she may lose rights to a large number of objects. In the case of NT and AFS, adding or removing a user from a shared group will not affect access control decisions until a user reauthenticates to the entire system. This allows NASD to take a lazy attitude towards reflecting group membership changes and having them take effect only when new access credentials, discussed in Chapter 4, are requested. However, in both NT and Appleshare, a group database is stored locally on the workstation. With AppleShare, the change is reflected the next time a user opens a file on the server, whereas NT does not reflect the change until the next time a user connects to the server. In the NASD world, in order for the change to take immediate effect, the file manager must explicitly, by modifying the object's metadata, revoke client's access to the affected objects.
- **Time of permissions check**: In AppleShare and NT, the statefulness of the protocol allows the fileserver to check access control permissions when a file is opened rather than on each individual operation. As a result, a client operation-in-progress will continue even if access control changes are made on the file. However, NASD is a stateless protocol, so the drive is unable to track "open" files. Normally, a filemanager attempts to provide a client with long-lived access credentials that allow the client to operate independently of the filemanager. However, in order to allow in-flight operations to complete, the NASD server needs to be kept informed of file opens and closes. If the server knows of the opens and closes, it can issue clients the appropriate access credentials to complete the in-progress operation even when the protection information on the file changes.

From the perspective of the drive, the most important issues, which ties together many of the concerns I have presented, are the number of sources of information necessary to make an access control decision and the complexity of making the decision. If group changes must have an immediate effect or an ACL change affects a group of objects, the NASD drive will either have more things to examine to make an access control decision or need to somehow update all the affected objects (if NASD is making local decisions only). If the filesystem has a rich access control scheme, it is much more difficult to use a simple abstraction such as groups to capture the expected behavior. Simpler, more structured schemes will be easier to implement in a commodity storage device. In Chapter 5, I will present a variety of types of access credentials which are partially motivated by these issues.

Chapter 4: The NASD Security System

As I described in Chapter 2, the high-level goals of the NASD security system are to:

- Enable applications to implement their application-specific policies over network attached storage devices.
- Protect the integrity of communication involving network attached storage.
- Deliver the scalability and aggregate bandwidth through the potential of the NASD architecture.
- Optionally, protect the privacy of communication involving network attached storage.

This chapter describes a basic security system, which is part of the NASD interface, for enabling a filemanager to implement its policies over storage. Initially, I present a high level view of the communications flow and iterate while adding increasing levels of detail. I describe the system in terms of properties that NASD requires from its cryptographic primitives. After presenting some detail, I show, based on published research, that my selections of primitives and their applications in NASD are secure. Finally, I describe the set of attacks that can be applied to NASD and list the mechanisms within NASD that prevent the attacks. In later chapters, I will refine this basic security system to improve performance and provide additional functionality from storage devices.

In order to control who accesses storage, the storage device must be able to conclude that a request came either from the filemanager or someone duly authorized by the filemanager. This allows the filemanager to control who can make requests to the storage and what requests are allowed. The complement must also be true: authorized clients must be able to recognize that a reply originated from the appropriate storage device. Optionally, the security system must be able to protect the privacy of data although the application layer can handle some of the privacy issues as discussed in Section 4.4.2.

Although the filemanager is not on the client-storage datapath, it must be able to assert its security policies with the same impact as if it were synchronously inspecting every request. However, in order to allow the filemanager to scale to large numbers of clients, I want the filemanager to not be involved in each and every request, i.e. to have asynchronous involvement in system operation. Asynchronous involvement allows the filemanager in a network attached storage architecture to scale by a factor of 2 to 5 more than a system in which the filemanager is synchronously involved [Gibson97a].

This chapter of my dissertation argues for a design of the security-specific portions of the NASD interface that enable a filemanager to be involved asynchronously with clients and still effectively have synchronous control over who accesses storage. This design has been implemented in the NASD prototype and used to implement both AFS and NFS filesystems on top of NASD [Gibson97b].

4.1 Basic Design

4.1.1 Overview

In this section, I give an overview of the NASD security system. I describe a pessimistic model of an adversary and then explain how the basic security structure in a server attached disk system evolves into a network attached storage system.

The system I will describe is designed to be secure when an adversary has full knowledge of the NASD interface, communication protocols, as well as the relevant cryptographic algorithms. The adversary may also pose as a legitimate client and gather information on the system through valid requests. The security of the system ultimately relies on the privacy of cryptographic keys, which will be discussed in Section 4.3, and not the secrecy of any of the protocols. I make no assumption about the underlying network providing any security guarantees. I make the worst case assumption, which is that an adversary can read, modify, insert, and delete arbitrary messages on the network. This is similar to an adversary having control of a critical router in front of the storage devices.

I also assume the client workstations are untrusted because an adversary may be able to control the operating system or software running on the workstation. A workstation may be in the office of someone who is attacking the system or the workstation may be in a low-security public workstation cluster. While the NASD system assumes that clients are untrusted, a user who choose to access NASD through a workstation is implicitly trusting the workstation not to abuse information the user provides. If a user accesses the storage system through a trojan horse client then she has compromised the security of the contents of any objects she has accessed through the trojan horse client.

In a server attached disk system, such as the distributed filesystems described in Chapter 2, the application server enforces its access policies directly by examining each request. A straightforward extension of this approach to NASD is for clients to ask the filemanager for permission to perform specific operations as shown in Figure 4-1. The filemanager understands the application-specific access control policies and makes an access control decision, which it “wraps up” and gives to the client in the form of an *access credential*. Any rights that are not explicitly granted in the access credential are

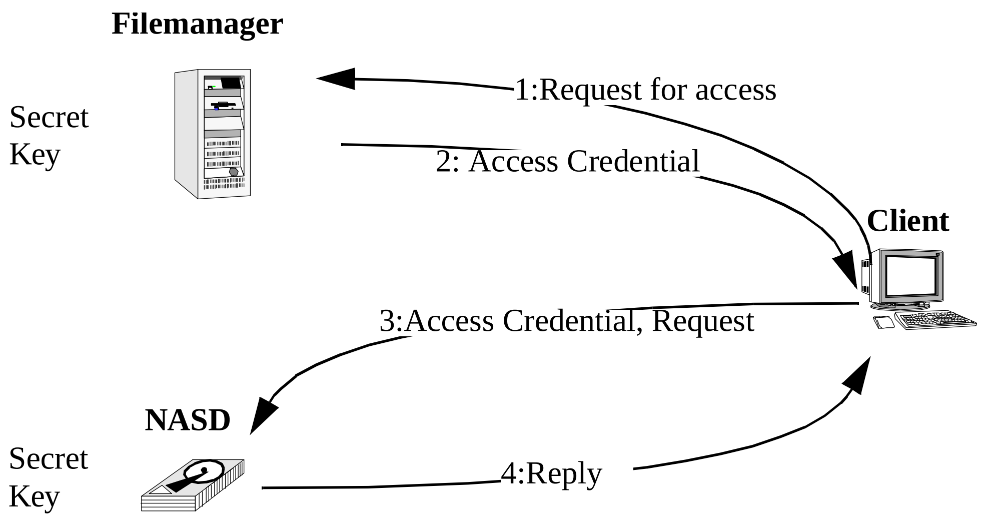


Figure 4-1 Flow of NASD Security

When a client wants to access a file stored on a NASD, the client first sends the filemanager a request for access rights (1). The filemanager then performs the application-specific access check to determine what rights should be granted to the client and sends the result, packaged in an access credential, to the client (2). The client sends the filemanager's access decision to the drive along with a request (3) which allows the drive to perform a simple check to properly enforce the filemanager's access decision before sending the client a reply (4).

forbidden to a client. The client then takes its access credential and shows it to the drive with each request as proof of the client's access privileges. If a client wants to perform an operation that is not permitted by the access credential, the client must request another access credential from the filemanager. However, the client can make repeated requests to the NASD with the same access credential as long as the requests only use the limited rights specified by the access credential.

This high level version of the NASD security system achieves the goal of asynchronous oversight because the filemanager is only consulted to issue the access credentials, i.e. render access control policy decisions. However, access credentials are examined by the drive on each request thereby giving the filemanager synchronous control. Access credentials are similar to classical capabilities [Dennis66] because they grant the bearer a specific set of access rights. Since the client sends the access credentials to the storage device on every request, the storage device does not need to maintain any long term state across requests which improves both scalability and fault tolerance of the device. The storage device can handle large numbers of active clients because an active client does not consume state on the device: only in-progress operations will consume state. Any storage device will have a limited number of requests per second that it can serve due to its CPU or media performance, which is a more natural constraint than having a connection-based approach where an active client in a potentially large site consumes state on the drive.

However, simply wrapping up a client's access rights and handing them to the client does not protect applications against a corrupt client that may modify the access credential. Without preventative measures, a client can take the access credential received from the filemanager and alter it to allow whatever rights it wants regardless of the filemanager's polices.

To address this problem, the filemanager signs the description of the client's access rights, which I call the *public access credential*, to prevent a client from modifying the access credential. I use HMAC-SHA1 [Bellare96a, NIST95] to generate an unforgeable signature of the public access credential, which I call the *private access credential*. Both are then sent to the storage device. The private access credential must be transmitted over a secure private channel while the public credential can be sent over a public channel. This separation of the credential into a public and private portion is similar to AFS's tokens. AFS tokens contain a secure token, which must be private because it contains cryptographic secrets, and a clear token, which does not need to be private because it does not contain any cryptographic secrets [Satyanarayanan89]. Taken together, the public and private access credentials are simply the access credential, and the flow remains the same as shown in Figure 4-1.

Since HMAC-SHA1 is a message authentication code (MAC) [Menezes98] that requires both issuer and verifier to share the same key, the filemanager and the storage device must share the cryptographic key used to generate the private access credentials. As long as the filemanager and storage device keep the key private, nobody else, including the clients, will be able to produce the private access credential corresponding to an arbitrary public access credential. This prevents unauthorized users from generating or modifying an access credential to obtain access to a storage device.

Unfortunately, a signature on the client's access privileges does not fully protect the system from an adversary that can modify network traffic. The adversary could change stored data, by modifying a write command, or give the appearance of different data being stored, by modifying a read reply, which would compromise system security. Therefore, the client and the storage device must include some form of signature, which is another HMAC-SHA1 message authentication code in my system, to prevent an adversary from modifying a request or reply as shown in Figure 4-2.

Since the client and storage device use a MAC to protect the integrity of communication, they must also share a secret key to use when generating a MAC. Clients use the private access credential, which the filemanager gave to the client as a signature over the public access credential, to bind a specific NASD request to the public portion of the access credential through a message authentication code, as well as similarly binding the reply from the storage device. This ties the MAC on the request directly back to the public credential, which describe a client's access rights, and allows the drive to readily verify the relationship between a client's rights and a request. The client includes the public access credential with each request, allowing the storage device — which shares a secret with the filemanager — to regenerate the private access credential which, in turn,

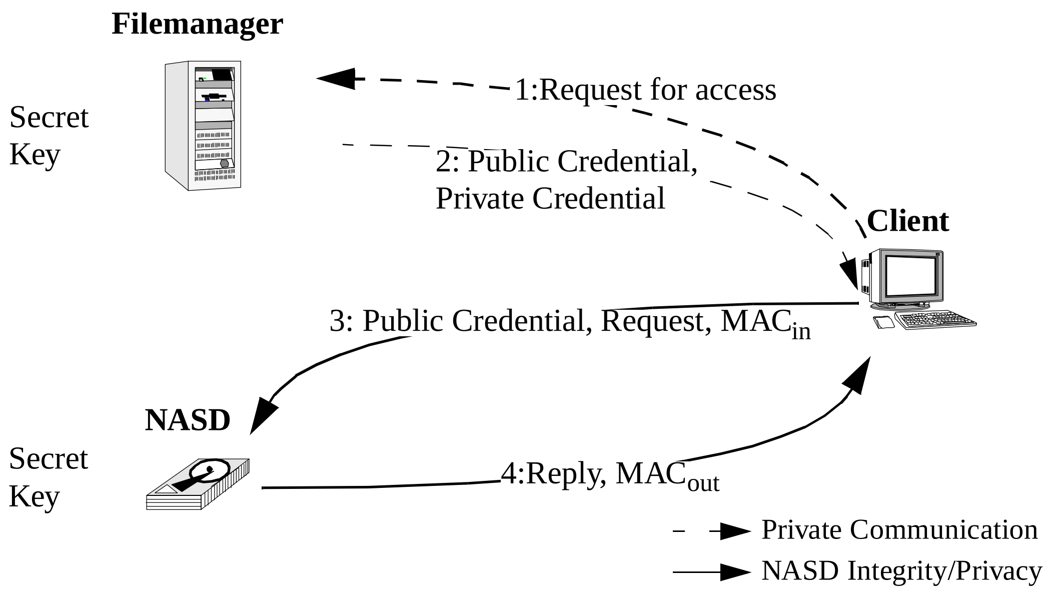


Figure 4-2 Overview of NASD Security

When a client wants to access a file stored on a NASD, the client first sends the filemanager a request for access rights (1). The filemanager then performs the application-specific access check to determine what rights should be granted to the client and sends the resulting description of access rights, the public credential, along with a signature of the rights, the private credential, to the client through an application-specific private communication protocol (2). The client sends public credential and a message authentication to the drive along with a request (3). The drive checks that the public credential grants appropriate rights and the MAC was generated using the proper private credential as a key which proves that the request came from a duly authorized client. Similarly, the reply includes a MAC which allows the client to verify the reply came from the NASD (4).

enables it to verify the MAC on the request. Additionally, the private credential can be used to protect the privacy of client-storage communication discussed in Section 4.4.4. An alternative is to simply encrypt the client-drive key under the filemanager-drive key and send this to the drive, which would be similar to the behavior of Kerberos. However, this requires encryption rather than a MAC and encryption is not only more expensive, but raises export issues.

Since the private credential is used to prove that a client was granted the rights described in the public credential, the private credential must be privately communicated to the client via an application-specific protocol to prevent an adversary from copying the entire access credential. The application defines how the information flows from the filemanager to the client but it must be kept private and is not part of the NASD interface because the drive is not directly involved. An application could use a customized private communication protocol to protect the privacy of the private credential or a standard protocol such as such as secure versions of Sun's ONC+ RPC [Sun97] or Transarc and OSF's DCE RPC [OSF91], all of which allow the application to hide data from observers.

This basic flow in the NASD security system delivers synchronous control of storage with asynchronous involvement of the filemanager, thus allowing the filemanager to handle a larger number of clients. By generating MACs of the public access credentials, the description of the client's access rights, the filemanager prevents a client from granting itself arbitrary rights. By including a MAC on each request and reply, the client and storage device guarantee the origin of their message exchanges. By requiring the access request to be private, an adversary is prevented from stealing a clients access rights.

Now that I have given a high-level description of the NASD security system, I will present an example and then go into greater detail about facets of the system.

4.1.2 An Example Read Request

In this section, I provide more detail on the NASD security system by explaining the function of access credentials and then applying them to a NASD read request. I also discuss the fundamental limitation of any access control system based on cryptographic tokens.

NASD access credentials are used to convince a drive that a client has been granted some specified set of access rights. They consist of two parts: a public portion — describing a clients rights — and a private portion — which is a cryptographic key. The private portion is a cryptographic key used to protect the integrity or privacy of communication between the client and storage device.

For each request, the client proves that it was authorized by presenting both the request specification and the public portion of an access credential to the drive as well as a message authentication code for the specific request. For now, assume that drive knows the unique secret key for all access credentials. In Section 4.2.1, I will discuss how this information is implicitly sent to the drive.

In Figure 4-3, I repeat Figure 4-2 at a greater level of detail to illustrate a client read request. The client sends a security header, public credential, read request, nonce, and a message authentication code to the drive which, together with the filemanager-drive key, provide all the information necessary for the drive to determine if a request is authorized. The security header, which is explained in detail in Section 4.4.1, and public credential together with the secret key provide the drive with sufficient information to generate the corresponding private portion of the access credential. Since only the filemanager and the drive have the secret key, they are the only ones who could have generated or distributed the appropriate private access credential thus preventing an adversary from spoofing or modifying the access credential. Using the private portion as a key, the drive generates a message authentication code of the received data and the nonce which it then compares to the MAC value received with the request. If the values match, the drive knows that the request came from someone who holds the private key corresponding to the public

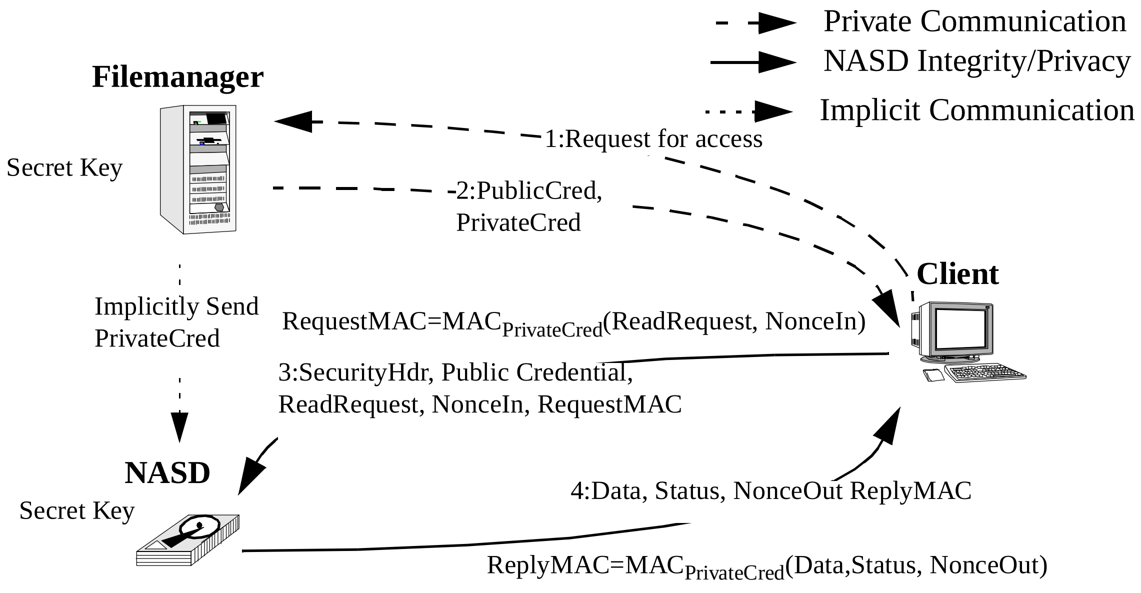

Private Communication
NASD Integrity/Privacy
Implicit Communication
Filemanager
Secret Key
1:Request for access
2:PublicCred, PrivateCred
Client
Implicitly Send PrivateCred
RequestMAC=MAC$_{PrivateCred}$(ReadRequest, NonceIn)
3:SecurityHdr, Public Credential, ReadRequest, NonceIn, RequestMAC
NASD
Secret Key
4:Data, Status, NonceOut ReplyMAC
ReplyMAC=MAC$_{PrivateCred}$(Data,Status, NonceOut)

Figure 4-3 Example Read Request

The client requests access from the filemanager (1) who provides the client with access credentials (2) using an application-specific private communication mechanism. The client generates a security header, which specifies how to process the request, and sends the header, the public credential, the read request, a nonce, as well as a MAC of the request and nonce to the drive (3). The drive uses the security header and public credential to generate the proper private credential which is used to verify the RequestMAC. After verifying the nonce is fresh, the drive generates a similar MAC on the reply and sends it to the client (4) which the client verifies similarly.

credentials sent with the request. From this, the drive can conclude that the request came from someone who was granted access by the filemanager.

In reality, this request could have been generated by a third party who was passed the access credentials by the original recipient. This is a limitation of any system based on cryptographic tokens, such as NASD’s access credentials, Kerberos tickets [Neuman94], or AFS tokens [Satyanarayanan89], since there is no way to tie the token back to the owner in an unforgeable manner. If the owner of a token gives the token to another user, the system has no way of verifying that the bits in the request came from the owner or someone to whom the owner gave the ability to impersonate himself. The only way to address this problem is by applying some secure trusted hardware which binds the cryptographic token to a specific piece of hardware, which is done both in Amoeba [Tanenbaum86] and in secure coprocessor applications [Yee95]. Although a third party could be issuing the operation making it impossible to know the actual request origin, if the original recipient of the access credential gave out the credentials then she is ultimately responsible for the operation which can be observed. Analogously, consider a key given to a manager for a safe containing valuable records. Although we cannot know who actually opened the safe, we know that only the manager or someone to whom the manager gave the key was able to access the safe unless the key was stolen or someone broke into the safe.

Nonces are included on every request and reply involving the storage device to protect freshness[1]. This allows the drive to verify that the request has arrived in timely fashion and has not been replayed by an adversary. Since the nonces protected by the MACs, an adversary is unable to modify a request to allow it to be replayed. NASD's nonces will be discussed later in Section 4.4.6.

4.2 NASD Access Credentials

4.2.1 The Private Credential

The private credential is a cryptographic key used by a client to bind a specific NASD request to the public portion of the credential through a message authentication code. This creates a strong relationship between the public credential, the request, a working key shared only by the filemanager and drive, and the message authentication code that the drive can verify for each request. Figure 4-4 illustrates this dependency relationship. The basis key is a shared secret between the filemanager and the drive, used with the public access credential to derive the private access credential. Since only the filemanager and the drive share this key, nobody else can generate the appropriate private access credential for a particular public access credential. The filemanager gives the private access credential only to an authorized client so a client holding the private access credentials implies that the filemanager has authorized the client with the rights specified in the public access credential. Since only someone with the private credential can generate the proper MAC on a request, if a drive receives the proper MAC then the drive can believe that the sender held the private credentials. Since holding the private credential implies the sender was authorized by the filemanager with the rights in the public credential, the proper MAC also implies that the requestor was granted the rights specified in the public credential.

If the drive generates the same MAC that was included in the request, the drive can be confident that the MAC, the public credential, and the request were all received without modification because a change in the request would change the value of the MAC and a change in the public credential would change the value of the private credential generated by the drive, which in turn would also change the MAC value.

1. A message is fresh if it was recently said by the apparent sender and has not been sent before.

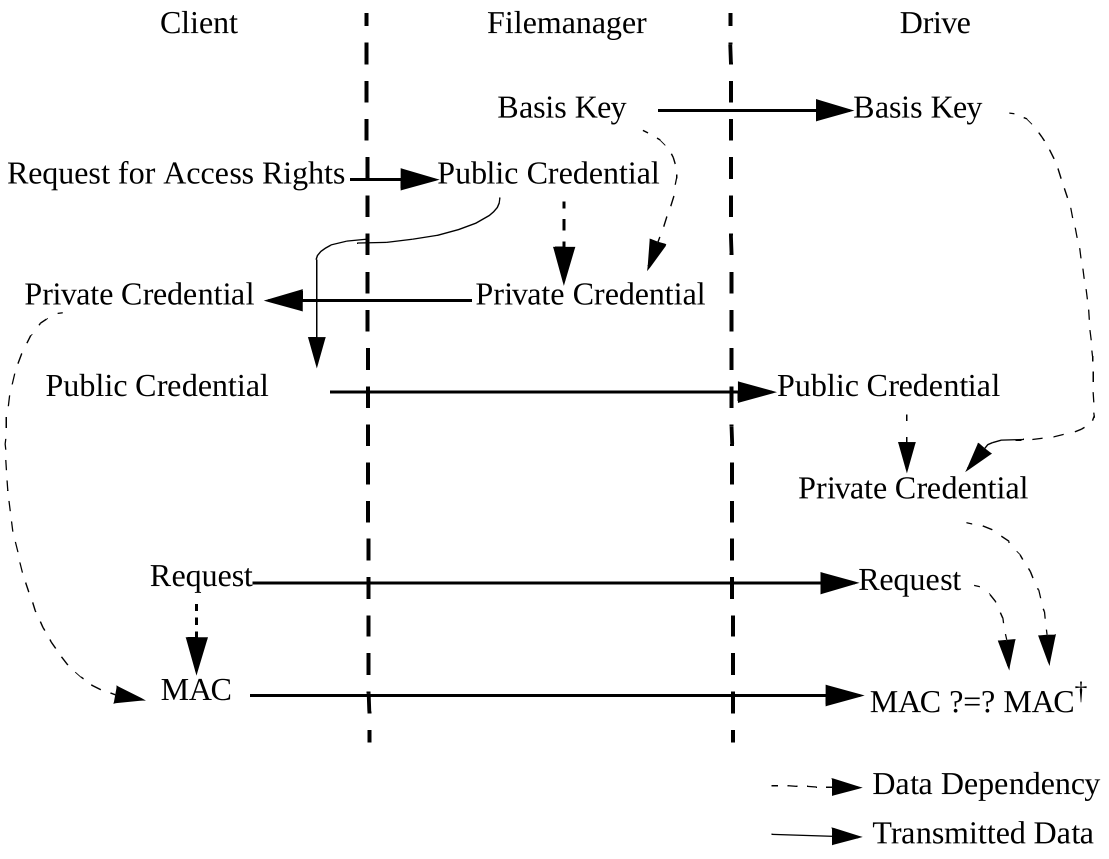


Figure 4-4 Message Authentication Code Dependency

Solid horizontal lines illustrate messages sent while vertical dashed lines indicate dependencies of one value on another. The dependencies and communication flow illustrate the steps required for a drive to be able to verify a request's MAC. The basis key is used to generate the private credential which is given to the client and used to generate the MAC that the client sends to the drive. Tracing the dependencies back from the final MAC calculation, the MAC depends on the basis key, which implies the filemanager has authorized the request.

In order for the preceding chain of logic to work, a strong relationship in the derivation of the private credential from the public credential and basis key is required. The private credential corresponding to a specific public credential is generated as:

$$PrivateCredential = F_{K_x}(PublicCredential)$$

where K_x is the basis key shared between the filemanager and the drive. What are the properties that are important for the function F? The function must be relatively easy to compute because the filemanager and the drive will need to evaluate F at least once per access credential. It must also have a large range, 90+ bits [Blaze96], and the output must be indistinguishable from random because the output will be used as a cryptographic key.

An adversary who is behaving like a normal client and collecting valid access credentials should not be able to use the valid credentials to generate their own credentials. More formally, given zero or more $(x, F_k(x))$ pairs, it is computationally infeasible to compute another pair $(x, F_k(x))$ for any new input $x \neq x_i$.

Finally, I would also like every bit of the input to affect every bit of the output i.e. a strong avalanche property [Webster85]. This property implies that a small change in the input will have an unpredictable impact on the output and allows $F_k(x)$ to safely generate keys when its input, the public credential, may change by only a few bits. Without strong avalanche property, a set of credentials where the public credentials differ in only a few bits might leak information about the key being used.

Essentially, NASD needs a keyed pseudo-random function [Menzenes97]. Modern iterated hash functions [Menzenes97] have, in practice, been shown to have considerable resistance to attacks and supply many of the properties that I require. Bellare et al. have used iterated hash functions as the basis for pseudo-random function families [Bellare96b]. Unlike many applications of pseudo-random functions, NASD does not evaluate *F* on a previous output of *F* in order to generate a sequence of random numbers. Instead, NASD uses a given seed exactly once to generate a single random value and does not evaluate *F* on results of previous results of evaluating *F*. This prevents attacks based on discovering the value of an intermediate chaining variable in an iterated hash function, a discovery that can be used to break the function [Kelsey97]. Additionally, an adversary can exercise only limited control over the contents of its public credentials so it is difficult for the adversary to mount a significant chosen plaintext on *F*. The difficulty of finding the key is the key recovery property commonly applied to message authentication codes and encryption algorithms, which suggests that a message authentication code based on an iterated hash may be appropriate for *F*.

In NASD, I instantiate *F* in NASD using HMAC-SHA1 [NIST95, Bellare96a], a message authentication code based on a cryptographic hash function. The SHA-1 hash function, in practice, provides all the desirable properties with the exception of the key dependency. Wrapping SHA-1 in the HMAC construction adds this dependency, creating a strong and efficient keyed pseudo-random function. This construction is similar to the Digital Signature Standard’s pseudo-random number generator [NIST94] although NASD uses a MAC rather than a hash function in order to provide the key dependency. The use of MACs to derive keys has also been proposed by Benjamin Reed et al. in the IBM SCARED project [Reed99] and by Mittra and Woo [Mittra97].

The cryptographic community has produced many hash functions which are provably equivalent in complexity to difficult number theory problems (such as the difficulty of factoring integers [Blum83] or the discrete logarithm problem [Goldreich86]), so their security can be more concretely described. Because the performance of the number theoretic hash functions is generally much worse than SHA-1, I selected SHA-1 over the number theoretic contenders.

Now, I will be a little more concrete about how an adversary could attack HMAC-SHA1 that is being used to generate the private credentials, i.e. generating keys. A malicious client can pretend to be a well-behaved client and repeatedly request access credentials from a filemanager. The client may be able to specify the contents of the access credential, which is a chosen plaintext attack, or may have no control over the contents, which is a known plaintext attack. Assuming a terabit network and access credential requests taking a modest 64 bytes, a client would be able to accumulate fewer than 2^{50} valid access credentials in 24-hours. This provides some intuitive sense of the amount of data an adversary could accumulate to mount an attack on the key generator and falls well short of the 2^{80} known text-MAC pairs expected to be necessary with a birthday paradox[1] attack to generate a forgery if HMAC-SHA1 is an ideal 160-bit MAC [Menezes98].

In the paper defining HMAC, Bellare et al. show that an attacker who can forge a MAC, i.e. generate a private access credential, can also break the underlying hash function, which is SHA-1. Currently, there are no known attacks on SHA-1 better than the brute force birthday attack so HMAC-SHA1 is believed secure. In the context of NASD, an attacker cannot generate her own private credential with the number of known text-MAC pairs that she could accumulate in a day.

4.2.2 The Public Credential

The public credential's primary purpose is to compactly communicate the rights granted to the client between the filemanager and drive. The public credential is given to the client who forwards it to the drive on each request. By passing the rights on each request, NASD pays a small fixed overhead on each request but the drive does not need to maintain any long-term record of the client's access rights. Each NASD request contains all the information required for NASD drives to grant access to an object except for the keys shared between the filemanager and drive.

The primary requirement for the public credential is to identify an object, or set of objects, and permissions on the object(s) that the filemanager grants to the bearer. The first component of this is to specify an object, which the initial NASD design does using the drive's unique identifier, the object's unique object identifier, the partition ID in which the object is located, and a version number included in the object's metadata. These fields together are called the *object specification*.

The version number in the metadata enables filemanagers to issue credentials tied to a specific version of the object. If the filemanager changes the version number, any access credential issued with the previous version number will be rejected by the drive. For example, a filemanager issues access credentials for version three of an object. At some

1. The birthday paradox is a standard statistical problem and a good explanation can be found in [Menzenes98]. The paradox is that the probability of two items in a uniformly distributed set drawn from n values having the same hash values is approximately $\frac{1}{\sqrt{n}}$.

time in the future, someone changes the permissions on the object to become more restrictive so the filemanager changes the version to four. A user with an access credential for version three will have her access attempts rejected until she obtains a new access credential specifying version four. This feature allows revocation on a per object basis in response to changes of the filesystem’s access control policies.

This approach to naming is very explicit and specific about what object can be accessed. A client must obtain an access credential for each and every object that it wants to access, which increases messaging. The limitation of this approach and a more powerful alternative to naming objects will be discussed in Chapter 5.

The second component of the public credential is to specify a client’s access rights. The NASD interface defines a set of basic operations such as GetAttribute, SetAttribute, ReadData, WriteData, Create, CreatePartition, FlushObjectFromCache, RemoveObject, and ChangeKey, which clients and filemanagers use to access storage. The storage device must be able to test that the access credential included with a request is valid for the requested operation. In the NASD interface, I implement *rights description* as a set of bit flags which are on for each operation that the public credential enables. As long as the number of operations is relatively small and well-defined, this implementation enables quick validity tests as well as access credentials that are valid for multiple different operations.

The object specification and the access rights description serve as the most basic and critical means of control that NASD must provide. However, it is not complete. Some applications, such as AFS on NASD, will want to grant access to only a portion of an object rather than an entire object. The NASD interface has extend-on-write semantics where a write beyond the end of an object extends the size of the object rather than returning an error. As a result, a client who can write to an object can consume arbitrary amounts of storage by simply writing bytes beyond the end of the file. AFS, which has a quota system, limits the amount of storage that a user can allocate. In order to enforce this quota when the filemanager is not involved in individual write requests, I included a byte-range field on each public access credential which allows a filemanager to specify what portion of a file a client may access. By specifying an uppermost byte that is within quota limits and escrowing the quota capacity, the byte range allows the AFS filemanager to enforce its quota policies [Gibson97b]. Other applications may also use the byteranges if multiple files or database tables are stored in a single object or if the application wants to grant access to sub-ranges of an object to enforce consistency guarantees.

The public credential must also contain enough information to allow the drive to construct the appropriate private credential. Therefore, the public credential must include an indication of the basis key to be used, as described in Section 4.2.1, to generate the private credentials which will be used to verify the request. If this key is changed then all outstanding access credentials issued using the previous key value will suddenly be invalidated.

This gives me a mechanism to revoke access: changing the version number of an object or changing the basis key. Since all access credentials include a cryptographic key, I want to limit their lifetimes in order to limit the exposure of the key. However, both of these revocation mechanisms force a large number of clients to recontact a filemanager to acquire fresh access credentials. I prefer to stagger the revocations when there is no security requirement for them to occur immediately. To allow a smoother influx of requests for reissued access credentials and to limit access credential lifetime, each access credential has a predefined expiration time after which it is no longer accepted by the drive. This allows the use of individual access credentials to be time-limited and requests for fresh access credentials to be spread over time just as the original requests were spread over time.

In NASD, there are a variety of different kinds of security options that can be used to protect client requests which allow the application to trade-off its security versus performance. Increased levels of security require more computation both at the client and the drive and can have substantial performance impact depending on the resources at the client and drive. These options describe different types of privacy or integrity protections that the clients and drives apply to messages and will be discussed in Section 4.4.2. The filemanager must specify which options are required for clients to use in order to use this access credential through the *minimum protection* field. This places a floor on the security that a client can apply to a request and have it accepted by the drive. A client can use additional protection options beyond the minimum required. Each partition on the drive also has a minimal set of protection options that the drive requires for any operation, these options must be included in the set specified in the public credential in order for the credential to be accepted by the drive.

In summary, the basic public access credential consists of:

- object specification — indicates which object the access credential is valid for
- rights description — describes what operations the access credential can perform
- bytes-range — constrains the access credential to being used on only part of an object
- basis key — tells the drive how to generate the proper private credential
- expiration — allows time-limited access credentials
- minimum protection — specifies the minimal amount of security a client must use with this credential

4.3 Keys

4.3.1 Key Hierarchy

In the preceding sections, I have referred to only a single basis key or secret key being shared between the filemanager and the drive. A single key is insufficient because it is not possible to conveniently balance the usage of a key versus the lifetime of the key nor can responsibility be delegated with a single long term key. Instead, NASD uses a small five layer key hierarchy that enables the storage owner to control access to storage. The bottommost layer of the hierarchy consist of the credential keys, which are used in client operations, while the upper layers of the hierarchy, collectively called the *administrative keys*, are primarily used for key generation and key management.

The number of long term keys in the system has been intentionally kept small in order to have well-defined roles for each key and reduce requirements for key storage. Key storage is different from other types of state with NASD because security ultimately rests upon the keys being kept private. However, greater measures should be taken to protect keys than normal data. By limiting this state, it becomes easier to store the keys in non-volatile storage and simplifies the task for filemanagers to control systems of hundreds or thousands of drives.

4.3.1.1 Master Key

For any storage device, there is ultimately some person or organization that is responsible for the device. I will call this person the drive’s owner. The drive’s owner has the *master key* for the drive and is able to use this to control it, primarily to manipulate the key hierarchy. The master key enables unrestricted access to the drive and is an immutable key. Thus, it should be used infrequently and be carefully protected from release. The master key is immutable because there is no higher level key with which to securely change it.

An exception to the immutability of the master key occurs when a drive is transferred to a new owner or administrative domain. When these situations occur, the master key is changed for the new owner and the entire storage device must be reinitialized to prevent the release of privileged information. An owner can send a reset message to the drive authorized by the master key which causes the drive to return all keys to the default factory settings and destroy all data stored on the drive. I note that some critics claim that because of magnetic memory effects, magnetic media can be subject to data recovery by skilled adversaries even when the data is erased and written over [Gutman96]. If the data is stored in unencrypted form, the data may always be recoverable until the media is physically destroyed.

An event in which the master key is compromised is treated as a disaster scenario and I consider the contents of a drive as lost. The drive must be reset, i.e. restored to its original state, and data recovered via RAID or a backup mechanism which has not had its integrity violated.

In all other cases, the master key is immutable and the owner must limit the amount of text encrypted or digested under the master key. This is achieved by only using the master key to delegate authority to a lesser key, the drive key, which can be changed by the master key.

4.3.1.2 Drive Keys

The owner may delegate authority to administer the space on the drive to an allocation manager by giving the allocation manager, a person or machine who actively manages the drive, the *drive key.* This key allows unrestricted access to the drive but, unlike the master key, can be changed if circumstances require. Examples of when the drive key may need to be changed are:

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

The drive key primarily allows the allocation manager to divide the drives capacity into distinct partitions and delegate authority for individual partitions to file managers. When a partition is created, the allocation manager uses the drive key to set a partition key that is then used to administer the partition.

4.3.1.3 Partition Keys

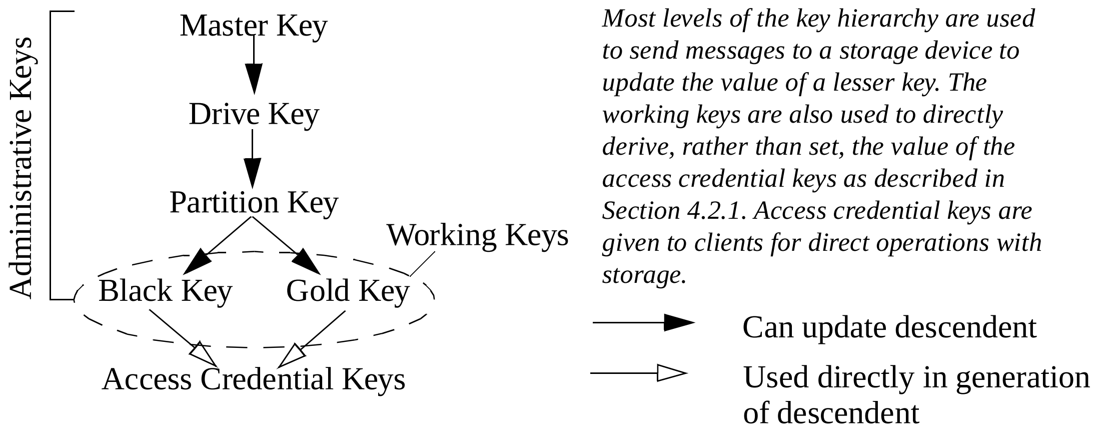
The allocation manager gives the filemanager a partition key for a new partition that the filemanager is granted the right to control. The only way the filemanager can change the partition key is by appealing to the allocation manager, who has the drive key. Therefore, the filemanager wants to limit the use of the partition key and extend its useful life by using it to set lesser keys, the working keys, which are used for regular drive operations. If a filemanager is compromised, the most important secrets it holds are partition keys and the adversary would gain access to only a single filemanager's partitions but *not* other partitions owned by other filemanagers on the same drive.

4.3.1.4 Working Keys

A filemanager uses the working keys to both generate access credentials and perform direct operations on a drive. For each partition, a NASD drive has both a "black" and a "gold" working key. By having multiple valid keys, NASD can smoothly transition from one key to another by issuing access credentials with one key while outstanding access credentials are allowed to expire rather than being synchronously invalidated. The working keys are use by the drive and filemanager to generate access credentials. If a working key changes, all access credentials created with it will immediately become invalid, which would cause a storm of requests for new access credentials. Using multiple keys avoids having a regularly scheduled key change operation revoke all outstanding access rights and avoids the request storm. Instead, a file manager can shift activity to a different key and allow most access credentials under the older key to gracefully expire over time before updating the older key.

Figure 4-5 NASD Key Hierarchy

Most levels of the key hierarchy are used to send messages to a storage device to update the value of a lesser key. The working keys are also used to directly derive, rather than set, the value of the access credential keys as described in Section 4.2.1. Access credential keys are given to clients for direct operations with storage.



As summarized in Figure 4-5, the master key is held by the owner but used only in emergencies. The master key can set the drive key which allows the allocation manager to partition the storage capacity and define partitions. The allocation manager uses the drive key to define partitions and partition keys which are given to filemanagers. The filemanagers use the partition key to set more frequently used working keys which the filemanager uses for both direct access and generating access credentials.

4.3.2 Details of Keys

4.3.2.1 Initialization of a NASD

When a drive is shipped from the factory, the owner needs some way of initializing it with key information so that only authorized users can subsequently exercise control over the drive. In this section, I describe a design to allow the owner to take control of a new storage device.

A drive arrives from the factory in an uninitialized state and it is moved into an initialized state by the new owner. I propose that the master and drive keys be initially set to the serial number of the drive, that no partitions exist, and that the drive is in an uninitialized state. Immediately on receipt of a drive, the owner sends the drive an initialize message which irreversibly moves the device into an initialized state, thereby solidifying the owner's control of the storage. The initialize message will also set the master and drive keys to prevent others from taking control of the storage device. In contrast, the partition and working keys are not set because they are tied to specific partitions which are not yet created.

The initialize message must be sent over a trusted channel, such as an isolated administrator's network, to avoid an adversary eavesdropping and learning the new keys. The trusted channel could be as simple as a single network link from an administrator's machine to the drive. If an adversary can eavesdrop on the initialize message, she will have the master key and unrestricted access to the drive, so I recommend that the drive be initialized while it is under the physical control of the administrator — for example, while the administrator can physically observe the integrity of the cable going from the drive to the administrator's notebook computer. No encryption or message digests are used because there are no shared secrets on which to base them; access to the drive in the uninitialized state is proof the owner is permitted to perform the initialization procedure.

Once a drive has been initialized, *the master key can not be changed*. In order to securely change a key if it becomes compromised, an owner needs a secure channel to the device. Over the network, a secure channel is equivalent to sharing a cryptographic key. In order to change the master key, another higher level key would be necessary. This is a recursive argument that will generate an infinite key hierarchy, which is clearly impractical. By defining the master key as the topmost key, I implicitly define a compromise of the master key as a disaster scenario where the device must be taken out of service. By design, the master is rarely used so its long lifetime will only produce a small number of messages for an adversary to use to attack the system.

4.3.2.2 Key Caching

NASD drives can cache access keys, reducing the latency of processing requests, by reusing the result of generating the private access credential. In order to maintain correctness, if someone changes a value on which a private access credential depends, the access credentials cache entry must be invalidated.

When a request arrives, the drive must take the public access credential, which was provided with the request, and generate the associated private access credential in order to verify the request. As discussed in Section 4.2.1, the private credential is generated by evaluating HMAC-SHA1 over the public credential. This requires computationally expensive evaluations of SHA1; the exact cost will vary depending on hardware or software support which is further discussed Chapter 7, and adds latency to each request.

Figure 4-6 illustrates the structure of a private credential cache. If the public credential is found in the cache, the stored private credential can be used rather than regenerating the private credential. This occurs when a client makes multiple requests with the same access credential within a small window of time. If the cryptographic operations are expensive, a cache hit allows the drive to shorten the length of the critical path to process a request.

Most workloads include some level of locality of access which can be exploited through caching to reduce the latency caused by key computation. In Section 5.4.5, I discuss the performance of an access credential cache using different object specifications. For the simple capability model that I am presenting in this chapter, 16 KB of memory achieves a hit rate of 40-50% on the two AFS workloads that I studied.

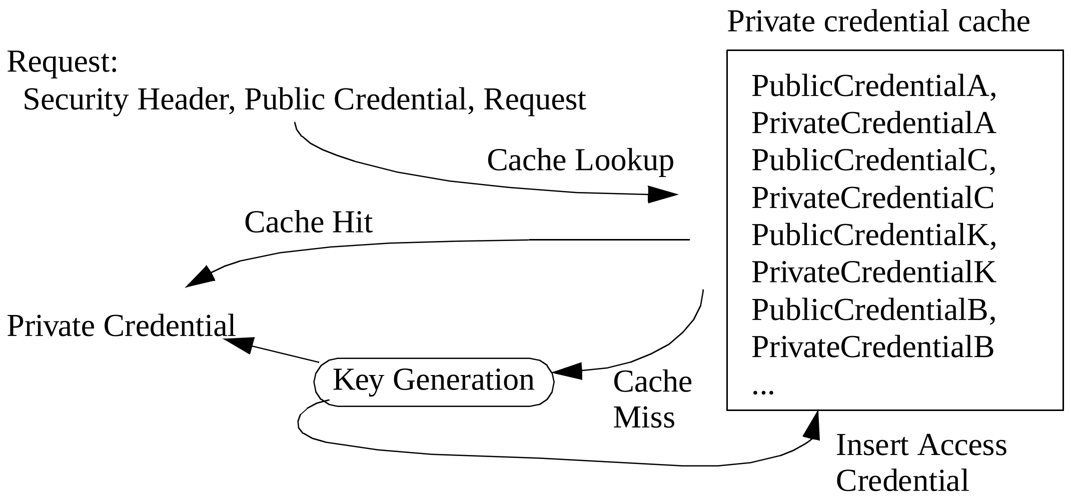

Private credential cache
Request:
Security Header, Public Credential, Request
Cache Lookup
Cache Hit
Private Credential
Key Generation
Cache Miss
Insert Access Credential
PublicCredentialA, PrivateCredentialA
PublicCredentialC, PrivateCredentialC
PublicCredentialK, PrivateCredentialK
PublicCredentialB, PrivateCredentialB
...

Figure 4-6 Private Credential Cache

This is a high-level view of how a NASD drive can cache private access credentials and avoid potentially expensive recalculation of keys. When a request arrives at the drive, the drive looks up the public credential in cache-memory perhaps using a hash table, to discover if it holds the appropriate private credential. If a cache entry is found, the drive can use the retrieved key to process the request. If no cache entry is found, the drive must recalculate the private credential and insert it in the cache.

An access credential in the cache becomes invalid when either its basis key changes or the associated object's access control version number (AV) is altered. Both of these values are included in the generation of a private credential, so cached private credentials using old values must be invalidated. If the cache entries were not invalidated, an old access credential could remain valid despite the filemanager having revoked it by changing the basis key or AV. The basis key changes through specific drive RPC which will also invalidate all affected cache entries. Since this is a rare operation, invalidation can be done with exhaustive search of the cache. In contrast, the AV may change regularly as permissions change or the filemanager uses it to support quota or consistency protocols [Gibson97b]. The AV changes when someone performs a set attribute, since the AV is part of the object attribute, and modifies the AV field. To maintain correctness in the cache, the old entries dependent on the previous value of the AV must be invalidated. Revocation due to AV changes can be implemented efficiently if the cache is indexed by the object identifier for the public credential.

4.3.2.1 Key Generation

In this section, I explain the difference between generation of an administrative key and generation of an access credential key. An important distinction between the creation of access credential keys and administrative keys is that access credential keys are defined by the NASD interface but administrative key generation is defined by the application. The generation of access credential keys is part of the NASD interface because the drive must understand how they are generated and the relationship they have with the access credential arguments in order for the drive to believe that the filemanager issued the acess credentials. In contrast, administrative keys are set by the administrators and have no special relationships to other arguments, so the administrators are free to generate them however they wish. However, I do offer some guidelines for system implementors.

Key generation and loading into the system is a *critical step* and the process should be carefully protected from an adversary. Administrators should generate the keys in the most secure environment available, preferably a trusted tamper-resistant device. If a tamper-resistant key-generating device is not available, administrators should minimally take precautions to ensure that nobody has modified the key generation software or the sources of randomness.

Since administrative keys are changed infrequently, the system designers have the luxury of time when generating a new random key. Given the choice between a slow but strong source of randomness and a much faster but less secure source, system designer should opt for the more secure option.

A new administrative key must be as random as possible. Ideally, some physical source of randomness such as radioactive decay or frequency instability in free running oscillators [Schneir96] will provide the necessary randomness. These approaches normally require special hardware and may be impractical in some applications. However, there are many potential sources of randomness in a computer system such as the low bits on the clock, keyboard latency [Zimmerman95], disk response characteristics [Jakobson98], behavior of various system timers [Lacy93], the low bits of space used in a filesystem, and other hard-to-predict phenomena. These sources of randomness can all be combined using a strong pseudo-random number generator (PRNG). If the pseudo-random number generator or sources of randomness are weak, attacking the PRNG could be the easiest attack against a system built on NASD. This proved to be the case in early versions of the Netscape browser where Berkeley graduate students noticed that the source of randomness to the pseudo-random number generator only had a few bits of randomness that could not be easily predicted and they were able to greatly reduce the size of the keyspace base on this observation [Goldberg96]. Since NASD relies heavily on a cryptographic hash function, multiple sources of randomness could be combined using the Digital Signature Standard’s pseudo-random number generator based on SHA-1 [NIST94], which is also the hash function used in the NASD prototype.

A new management key must *not* be directly derived from the previous version of the key. The goal of changing the keys is twofold: I want to limit the amount of encryption or MACs generated with a key and force an attacker to attack different versions of a key separately rather than being able to leverage breaking a past key to break a current key. If an adversary compromises version k of a key, the adversary cannot use this to help break version $k+1$ since version k was not used to set version $k+1$.

An adversary who compromises a higher level key can use the compromised key to set a lower key or eavesdrop valid settings of lower level keys. So, higher level keys are designed to be used much less frequently than lower level key. This limits the number of cryptographic operations performed under the longer-lived keys, which makes the task of breaking the keys more difficult for an adversary.

It is worthwhile to note a drive is *never* required to generate random values. Since only the administrative machines use a random number generator, only they need to be updated in the event that a flaw is found in the random number generation or when a better source of randomness becomes available.

4.3.2.2 Key Length and Use

In this section, I discuss how NASD uses the output of HMAC-SHA1, which generates access credential keys as discussed in Section 4.2.1, to generate subkeys for both MACing and encryption. If either the MAC key or encryption key is compromised, the other operation must remain secure, and at least be broken independent. Therefore, I cannot use the same key for both operations. There are three approaches to generating subkeys from the 160-bit output of HMAC-SHA1: divide the bits, use an alternative to HMAC-SHA1 with a larger range, and hash the 160 bits with unique constants.

The simplest solution would involve splitting the 160 bits into two 80-bit keys. Unfortunately, these keys are smaller than the 90 bits recently recommended by a panel of eminent cryptographers for long-term security [Blaze96]. The split could also be asymmetric, for example 70 bits for a MAC key and 90 bits for an encryption key, because a MAC only needs to remain unbroken for the duration of the access credential, since it will be worthless after the access credential expires, while data should remain private forever. The shorter MAC key raises concern that the key may be too short and Moore’s law may advance computing so that the MAC key could be broken before the access credential becomes invalid.

Some function other than a single evaluation of HMAC-SHA1 could be used to generate keys. This could be done by either using an *F* function that produced longer values, although there are no obvious strong candidates, or using two distinct keys for HMAC-SHA1 in order to generate two distinct outputs. This requires that the file manager send longer keys to the client, which may not carry a substantial penalty, but may also require that more key bits be stored at the drive, which could be expensive.

The 160-bit output of HMAC-SHA1 can be hashed again with two unique constants to generate the MAC and encryption keys. I can provide the full 160 bits of randomness to each key by padding out the 160 bit *Key0* to a 512 bit block using two different constants and hashing each of these padded blocks with SHA-1 to produce two new 160-bit keys, *KeyA* and *KeyB*. Both *KeyA* and *KeyB* share *the same* 160-bits of randomness but, since SHA-1 is a one way function, an adversary cannot use knowledge of one key to derive the other key unless she can invert SHA-1. An adversary who can break *KeyA* will not be able to use this achievement to break *KeyB* or vice versa. Computing these derived keys is a minor performance hit at the drive but the cost can be amortized across multiple requests through key caching as I mentioned in Section 4.3.2.2.

Of the three alternatives presented, merely splitting up the bits is the largest security risk. An alternative to HMAC-SHA1 with a larger range would be appropriate but there is no obvious function that meets the criteria described in Section 4.2.1 and has a larger range. Therefore, I hash the 160 bits with unique constants to generate the MAC and encryption keys.

4.4 Design Details

In this section, I describe the details of how the NASD security system protects the integrity, privacy, and freshness of NASD requests. I start by explaining how a drive identifies what cryptographic key to use to process the request and then I describe the various types of protection that NASD can provide as well as the cryptographic operations necessary to provide the protection. Finally, I discuss how NASD protects the freshness of requests.

4.4.1 Security Header

The security header gives the drive enough information to identify both the proper key to use to process the request and the cryptographic operations to perform on the request. The security header is always sent in plaintext so this information will always be available to an adversary. Therefore, only the minimal required information is included in the security header.

The first function of the security header is to identify which key to use to generate the private credentials. The public credential may be encrypted, which is discussed in Section 4.4.2, so the security header must include enough information so that the drive can decrypt an encrypted public credential. This motivates the security header to have the following fields:

- **type**: The type tells the drive which key in the key hierarchy to use to process the request and also tells the drive if this request uses an access credential or is directly authorized by an administrative key.
- **partition**: In order to distinguish between the multiple partition, black, and gold keys in a drive’s key hierarchy, the security header also includes a partition field. For example, the type field may indicate that the request uses an access credential created with a black working key but the partition indicates which of the black working keys are used.

Once the drive knows which key to use, the drive needs to know what to do with the key, i. e. how to process the request. This motivates the final field of the security header:

- **actual protection**: The actual protection field tells the drive how to undo or verify the protections that were applied to the request by the filemanager or client. The different protection options are discussed in the next section.

4.4.2 Protection Options

Applications built on NASD will require different security properties depending on concerns such as the environment in which an application operates, the cryptographic performance available from clients and storage devices, and the security requirements for an application. In this section, I describe the different options that an application and client can select from in order to balance these concerns and how the security options are determined for a request.

4.4.2.1 Description of Options

In many cases, applying security guarantees such as privacy or integrity will be computationally expensive if drives or clients lack the capability for high throughput cryptographic operations. Ideally, any message exchanges with a drive would have their integrity and privacy protected from an adversary. Realistically, different clients and drives will have different capabilities and there will be different application requirements. Therefore, I allow the filemanager to determine the minimal acceptable levels of protection and trade off the performance concerns versus security concerns.

NASD requests and replies, ignoring the security-related fields, are divided into two components: data and arguments. The data is the potentially large sequence of bytes transferred as a result of read or write operations. All other information falls into the category of arguments. This includes information such as the object which the operation access, object offsets, and return codes.

The NASD security has the following security options that can be set on a per access credential basis:

- **IntegrityArgs**: The integrity of the arguments and the nonce is protected by a MAC.
- **IntegrityData**: The integrity of the data is protected by a MAC.
- **PrivacyArgs**: The privacy of the argument is protected with encryption.
- **PrivacyData**: The privacy of the data is protected with encryption.
- **PrivacyCredential**: The privacy of the public access credential is protected with encryption. Protecting the privacy of credentials makes it more difficult for an adversary to track a client’s access patterns because only the security header, discussed in Section 4.4.1, is available in plaintext to the adversary.

4.4.2.2 Sources of Security Options

When a request arrives at a drive, there are three sources of protection options that are involved. For every partition, the filemanager sets a floor on the set of required security options for any operation on that partition. For meaningful security, the filemanager should require *at least* **IntegrityArgs** on the partition. Without this minimum, an adversary could generate a request with the security header and access credential indicating no security and then issue arbitrary requests to the drive. If only **PrivacyArgs** is required on a partition, an adversary could send random requests to the drive, which would be decrypted and subsequently some would be accepted by the drive with unpredictable results.

In order for the filemanager to control on a per-access credential basis the security protections used, each access credential includes a minimum set of security options that must be used with the credential. This enables a filemanager to apply different degrees of security to different objects within a partition.

A client may want to apply greater security than the minimum required by the filemanager. A client can set the security options used in the security header, described in Section 4.4.1, while both the partition and access credential values are set exclusively by the filemanager. For example, this allows a client to use private communication in a system that may only require integrity.

When a request arrives, the drive first examines the security header to determine what options are used on the request. The header informs the drive if **PrivacyCredential** was used to hide the contents of the credential. If the credential was encrypted, the remainder of the security header defines which key to use in decrypting the credential. At this point, the drive verifies that the security header's listed security options include all the options required by both the access credential and the partition minimums.

All three of sources of security options are used by the drive to verify each request. They are strictly ordered: the access credential options must be superset of the partition options and the security header options must be a superset of the access credential options. If this ordering fails, the request will be rejected by the drive because someone's decision about the security options failed to comply with a higher-precedence decision.

4.4.2.3 Attacking the Security Options

The security header is neither encrypted nor directly protected with a MAC, so an adversary can modify the security header's security options field. In this section, I explore what the adversary can achieve by adding or removing options from the security header.

The adversary can add security options to the security header. If the adversary adds integrity protection, the drive will attempt to verify a non-existent MAC. This attack is essentially the same as an adversary randomly modifying a bit in the request causing the request to fail. Because the network is unsecure, an adversary can always modify bits in a valid request and cause the request to be rejected by the drive. If an adversary adds a privacy option, the drive will decrypt data that is unencrypted, which will fail the integrity check if an integrity option is also used. If no integrity option is used, the adversary can already modify the data so the essentially random manipulation of decrypting unencrypted data is not a very useful attack.

The adversary can also remove security options from the security header. If the adversary strips off an integrity option then the adversary can freely modify the request. If the access credential or partition specify that the integrity protection must be used then the request will be rejected because it lacks the protection. If the filemanager wants to protect integrity, the filemanager must set that minimum on the partition or in the access credential. If the adversary strips off an encryption option then the drive will attempt to store encrypted data directly without decryption. However, if the corresponding integrity option is used, **IntegrityArgs** for arguments and **IntegrityData** for data, and required by the partition minimum, then an integrity check will prevent this attack.

4.4.2.4 Minimal Protection of Drive Management Operations

There are some special operations on the drive that have operation-specific minimal security requirements. In order to manipulate drive configuration information, a request must be protected with at least **IntegrityArgs** and **IntegrityData** to guarantee that critical configuration parameters are never changed to a corrupted value. Key management operations require **PrivacyData** to prevent an adversary from observing the new key values. Without these minimal requirements, an adversary could easily change configuration options or snoop keys, which would allow the adversary to hijack the drive and have unrestricted access to the data.

4.4.3 Audit Logs

Some applications require the ability to audit client operations on storage. The purpose of the auditing mechanism is [NCSC87]:

- The audit mechanism must allow review of patterns of access.
- The audit mechanism must allow discovery of internal and external attempts to bypass the protection mechanism.
- The audit mechanism must deter attempts to bypass system security.
- The audit mechanism must provide additional assurance that attempts to bypass security are recorded and discovered.

If auditing is desired in a NASD implementation, the drive needs to bind requests back to the source of the request. However, NASD’s capabilities are not tied to a specific identity so additional information must be included in each access credential to facilitate audit trails. The drive can simply store the access credential for each request with the audit record for the request and require the application to reconcile the access credential's contents and the recipient of the access credential. This requires that the filemanager maintain a log of all access credentials to enable this reconciliation. If a filemanager is servicing high numbers of clients and drives, such a log will grow larger, so I want to eliminate this bookkeeping overhead.

Instead of logging the entire access credential, each access credential includes an uninterpreted identifier, called the *audit ID*, which is recorded in any log entries. The drive does not understand the contents of the audit ID beyond the fact that it is recorded in any log entries. The filemanager defines the semantics of the audit ID and can use it to encode information such as a user identity, process group, or role identity which will be stored on each request. By encoding the relevant information in the audit ID field, the filemanager avoids recording all access credentials and simply records the mapping of audit ID to an application-specific security concept.

When an application runs in a server attached system, the fileserver is able to serialize all requests and form a single audit log for multiple storage devices. In a network attached storage system, the application needs a mechanism to combine multiple audit logs from the storage devices.

Each storage device has a unique time value that advances at a constant rate, which will be explained in Section 4.4.6.3, that can be used to label individual audit log entries. The filemanager must maintain roughly current values of each drive's clock in order to generate valid requests and set expiration times on access credentials. The relationship between the clocks of multiple drives can be recorded by the filemanager so, at a later time, the filemanager can reconcile the logs of multiple storage devices into a single log. Since the accuracy of the clocks can vary, log entries on different drives may be reordered but will be within the error bounds of clock protocol.

Auditing can have a significant impact on system performance regardless of whether the logs are stored locally or remotely. If audit trails are stored locally, logging all operations reduces the I/O bandwidth available to clients and consumes valuable storage. Seagate's experience with logging summary environmental data using their S.M.A.R.T. feature has shown that logs can have significant performance impact [Seagate98]. Seagate is also concerned that an audit system generating an extra message per I/O could be a significant performance problem in a transaction system [Seagate98]. However, filesystem tracing research has shown that distributed logging, which batches together log records at log-clients, i.e. the drive, and periodically sends the results to a centralized server, can have less than a 7% performance degradation on filesystem benchmarks [Mummert94]. If each individual client request generates a unique audit message to a remote server, system performance will be worse than if the messages are batched together. Unfortunately, batching together audit records before committing them to the remote audit log server opens a window during which audit records will not be processed. This illustrates that the choice of auditing mechanisms needs to trade off the performance of the system against the auditing requirements of the application.

4.4.4 Cryptographic Primitives

In designing the NASD security system, I've favored the use of message authentication codes (MAC) as the basic primitive of security in NASD over encryption whenever possible. This bias has produced a system where most of the security guarantees and key distribution can be provided without using encryption. This is desirable because encryption algorithms are generally slower than MACs and encryption algorithms are subject to U. S. export restrictions, while MACs have no export restrictions.

For any implementation of the basic NASD security system, a designer could use any of a wide variety of available MACs or encryption functions [Menzenes98] but both algorithms should allow efficient implementation in both software and hardware. Some drives and clients may have hardware support for cryptography, which I will discuss in Chapter 7, while others will rely on a general purpose or embedded processor.

Licensing restrictions of the cryptographic algorithms must also be factored into the decision of which algorithm to use. U. S. government standards are generally free of any restrictions and are endorsed by the U.S. government so they are appealing options. It is my hope that endorsement by the U.S. government implies that the algorithms have undergone a reasonably rigorous evaluation by the experts at the National Security Agency and other federal agencies in addition to the public review of the algorithms.

Next, I present arguments for the use of HMAC-SHA1 and Triple-DES in NASD, followed by a discussion of using Triple-DES in counter mode to enhance parallelism in the encryption/decryption.

4.4.4.1 Message Authentication Code

Message authentication codes are used to both generate the private access credentials, as I discussed in Section 4.2.1, and for protecting the integrity of communications. When used to protect communications, the MAC provides data origin authentication which is defined as: "... a party is corroborated as the (original) source of the specified data at some (typically unspecified) time in the past" [Menezes98]. This provides two basic assurances: identification of the source of the data and data integrity. If a message recipient generates the same MAC as she received with a message, she can conclude that the source of the data was the holder of the appropriate key, the private credential, and that the data was not modified in transit. In this section, I present the requirements that my design places on the MAC and argue that HMAC-SHA1 is a good MAC for NASD.

For key generation and integrity protection, the key of the MAC must be large enough prevent brute force attacks on the keyspace. A panel of cryptographers determined that 90+ bits is the current minimal key length for long-term security of encrypted data [Blaze96]. This is a good lower bound for a MAC key length because encrypted data must remain encrypted for longer periods of time than a MAC must remain unforgeable.

The range of the output of the MAC must be large enough to make it computationally infeasible to find a collision based on the birthday paradox. Finally, the MAC must not leak useful information about the key through its output, i.e. it must prevent key recovery from other information. These are standard requirements for a message authentication code [Menezes98].

For my research, I adopted HMAC-SHA1 as the basic message authentication code algorithm [Bellare96a] based on the SHA-1 hash function [NIST95]. SHA-1 is used as part of the U.S. Digital Signature standard [NIST94] and has, to date, passed the scrutiny of review both internal and external to the government. Unfortunately, SHA-1 was developed in secrecy by the National Security Agency so the exact design criteria is not public. As a result, some people still harbor concerns that the government has deliberately introduced a flaw into SHA-1, although no evidence has been found to support this belief. Bellare et al. have shown how to relate the security of HMAC-SHA1 back to the security of SHA-1 which provides a strong grounding of the strength of the HMAC construction [Bellare96a]. HMAC-SHA1 has also been proposed as an IETF standard and it is used as one of the defaults in early IPsec implementations [Madson98]. HMAC-SHA11 processes a key of up to 160 bits and an arbitrary length input in 64 byte blocks and produces a 160 bit result which is large enough for both key generation and use as a normal MAC. Finally, HMAC-SHA1 is free of licensing restrictions which make it easy to redistribute. Together, these issues were a compelling argument to use HMAC-SHA1 in my prototype design.

To be a little more concrete about the difficulty of breaking HMAC-SHA1, consider how much information an adversary can collect to attack if HMAC-SHA1 is being used to protect the integrity of communication. An adversary can watch valid requests that pass over the network and collect message-MAC pairs and then attempt to forge a MAC. If I assume a terabit network and 64 byte operations, an adversary can accumulate fewer than 2^{50} valid known text-MAC examples in a 24 hour period which is well short of the 2^{80} known text-MAC pairs required to generate a forgery using a birthday attack under the assumption HMAC-SHA1 is an ideal 160-bit MAC [Menezes98].

An earlier version of the NASD prototype used HMAC-MD5 to provide integrity. However, work by Dobbertin on the MD5 compression function created doubt in the cryptographic community about the strength of MD5 and it was recommended that MD5 not be used if collision resistance is required [Dobbertin96]. HMAC-MD5 is still a safe application of MD5 but one of the optimizations I investigate in Chapter 6 relies on the underlying hash function used with HMAC being collision resistant so I moved from an MD5-based system to an SHA-1-based system.

4.4.4.2 Encryption

Encryption is the basic tool used to provide private communication, i.e. prevent an unauthorized adversary from learning the contents of some communication. The NASD security system uses symmetric key cryptography because operations with the alternative,

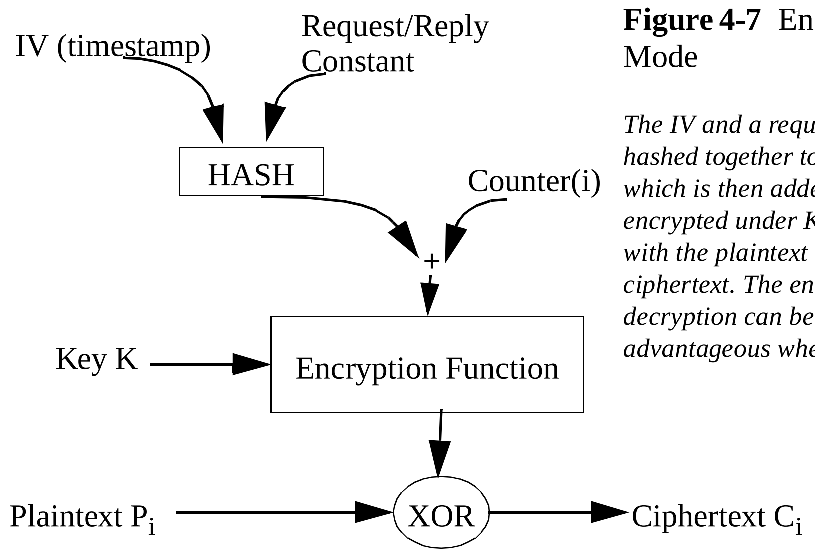


Figure 4-7 Encryption in Counter Mode

The IV and a request/reply constant are hashed together to generate a random IV which is then added to the counter and encrypted under K. The output is XOR'd with the plaintext to generate the ciphertext. The encryption and decryption can be parallelized which is advantageous when done in hardware.

public key encryption, are several orders of magnitude slower than private key operations [Menezes97].

The NASD prototype uses 2-key (112 bit) EDE Triple-DES to provide privacy protection. This mode of Triple-DES is an ANSI standard [ANSI85] and was recently proposed as an official interim government standard to replace single DES [NIST99]. Triple-DES is considered perhaps twice as secure as single DES, which is unfortunately rather insecure, [1] due to its key length which is twice that of single DES. Substantial public review of Triple-DES has not produced any significant attacks. As an additional advantage, Triple-DES can be easily implemented using off-the-shelf DES logic cores or chips which have been developed over DES's long lifetime. I intend on migrating NASD to the Advanced Encryption Standard (AES) when it is finalized [NIST98], although this may not be for several years.

4.4.4.3 Encryption Mode

NASD uses the cipher in counter mode, shown in Figure 4-7, which is similar to the better known output feedback mode [Diffie79], because counter mode enables block-level parallelism in the encryption/decryption processing. Counter mode operates by encrypting a counter, which identifies the block's location in the message and the output of a hash function in NASD, then XORing the counter with the plaintext. Since the encrypted value of a plaintext block depends only on the counter and key value and not the preceding data blocks (which is true in the more commonly used cipher-block-chaining mode [Menezes98]), each encrypted block can be computed independently and in parallel. Decryption behaves in a similar manner.

1. A message encrypted with a 56-bit key with single DES was recently decrypted in 23 hours by an ad-hoc group of people on the Internet along with EFF's Deep Crack machine in response to RSA's DES-III challenge [EFF99].

The security of encrypting in counter mode relies on the strength of the underlying encryption algorithm and the uniqueness of the initial vectors used to seed the counter. If two messages are generated with the same key and initial vector, an adversary can XOR the two requests together and discover the messages. Conveniently in NASD, the client provides the system with a unique value for each request -- the timestamp. However, the timestamp cannot be used on both the request and reply without revealing the messages.

I transform the NASD timestamps into a pair of unique IVs by concatenating the timestamp with a reply or request constant and then hashing the result. For each direction, request or reply, I define a constant which is concatenated to the timestamp. Since I am using Triple-DES which uses 64 bit blocks, I need a 64 bit IV. However, NASD timestamps are 64 bits so the concatenation of timestamp and constant is too large. To reduce the size, I hash the timestamp and constant with SHA-1 and take the lower 64-bits. I am using the timestamp and constant to seed a public random function, approximated by SHA-1, which generates the initial vectors. Since SHA-1 behaves like a random function, an adversary will be unable to find a combination of a reply and a request that will have the same IV even if the adversary has full knowledge of SHA-1, the timestamps, and the constants.

The drive can also precompute the encrypted counters as soon as the drive identifies the proper key to be used. For example, if a drive receives the first half of a request and then experiences a network delay. The drive can use the delay time to generate the encrypted counters to speed decryption on the second half of the request. When the second half of the data is received, the drive will be able to process it with less effort.

Recent work by Bellare et al. has shown that an adversary has no advantage over random guessing when using an ideal cipher in counter mode [Bellare97a]. This provides a strong basis to believe that, despite being an uncommon mode, counter mode is secure.

To put the security of Triple-DES into perspective, consider how much chosen-plaintext an adversary can accumulate in 24 hours to attack a single key. Assuming a terabit per second network, the client can generate 2^{50} chosen-plaintexts. There is no known attack to break Triple-DES with this amount of chosen-plaintext. Exhaustive key search is still the most efficient attack which requires approximately 2^{112} Triple-DES calculations.

4.4.5 Privacy and Integrity Together

A client and drive may protect both the privacy and integrity of an operation. Providing both privacy and integrity requires that a message be encrypted and have a MAC generated. This raises the question of how to order the cryptographic operations. I can MAC the encrypted data or MAC the plaintext data.

Generating a MAC of the encrypted data is appealing because it optimizes for the receiver’s role. Message are sent as:

$$E(data),MAC(E(data))$$

Since both decryption and MAC occur on encrypted data, the receiver verifies the MAC in parallel to decrypting the data. In most communication protocols, the role of the receiver is more difficult than the sender because the receiver has less control and is reacting to the sender rather than driving the communication. Even though the protocol stack at either end may be essentially identical, the receiver must handle demultiplexing, memory management, and generating interrupts (or otherwise signalling packet arrival). Thus, I would like to simplify the role of the drive when it receives requests.

Unfortunately, MACing encrypted data introduces a flaw because an adversary can tamper with the protection option field of the security header and reduce the security from privacy and integrity to simply integrity. The MAC generated over the encrypted data, now being treated as plaintext, would still be valid but the data would be treated as if it were unencrypted. In a write operation, an adversary could remove the privacy protection from an in-flight operation and force the drive to write encrypted data rather than plaintext thus compromising the integrity protection.

In order to protect the integrity of data stored on a NASD, clients and drives must generate the MAC of plaintext data and then encrypt the results. In this case, message are sent as

$$E(data), MAC(data)$$

The sender can encrypt the data and generate the MAC in parallel but the receiver must decrypt the data and then verify the MAC. This places more of a burden on the receiver but it does not have the flaw I described of MACing encrypted data. Additionally, MACing the plaintext rather than encrypted data permits an optimization of the MAC which I will describe in Chapter 6.

4.4.6 Freshness

Regardless of the privacy and integrity features provided by the NASD security system, NASD drives and clients must be able to detect replayed or delayed messages, i.e. they must be able to verify the freshness of a request. An adversary must not be able to record or intercept requests made to a NASD drive and replay the requests at another time. If an adversary can replay requests that the drive will accept then an adversary can modify stored data and force the drive to perform unauthorized operations.

The replay attack is not as powerful as forging arbitrary requests but can still cause serious damage to the integrity of a file system or database. For example, an accounting system storing prepaid account data could be modified by an adversary replaying a write operation that increased their account balance and consequently decreased revenues for the system operators.

To prevent a replay attack, the drive must be able to verify that it has not already seen the request. To prevent a delay attack, the drive must be able to verify that the request was recently sent to the drive. These two problems are normally addressed using a field called the nonce which allows the drive to verify the freshness of the operation. There are three alternatives to use for a nonce: random nonces, sequence numbers, or timestamps.

4.4.6.1 Random Nonces

The ideal protection against a replay attack would be for each request to include a random unique identifier generated by the sender, called a random nonce, as part of the operation. By comparing the nonce in an operation against all past nonces, the drive could verify that the operation is not being replayed. Unfortunately, maintaining a record of all past requests is not a feasible approach because of the limited storage available on a NASD drive.

4.4.6.2 Sequence Numbers

Sequence numbers can also be used to protect the freshness of requests and require less space than random nonces. In a connection-based environment, it is trivial to add a few more bits with connection state to provide a sequence numbers. Indeed, most connection-based protocols already incorporate some form of sequence number. However, denial of service attacks similar to SYN flooding are often possible in the connection set-up phase [Schuba97]. This type of denial of service attack is caused when an adversary takes advantage of a bootstrap phase, such as allocating sequence numbers or connection state, to force the receiver to allocate resources on behalf of the adversary. Swamping the receiver, e.g. the storage device, with messages that allocate state can force the receiver to run out of space for the state and reduce performance or crash the receiver. In a connectionless application like NASD, a sequence number requires an extra message exchange with the drive to initialize the sequence number, which adds latency to initial client requests and requires the drive to maintain the current sequence number, i.e. connection state.

For each client, the drive needs to maintain at least the next sequence number and perhaps a list of other expected sequence numbers. If requests can be reordered on the network, the drive will need to maintain a list of unreceived yet still expected sequence numbers. Deciding how much memory to allocate to these sequence records requires a designer to balance the number of active clients with the amount of reordering of requests that the drive is willing to tolerate.

4.4.6.3 Timestamps

Timestamps are also a good mechanism for preventing replay or delay attacks. Timestamps have been criticized for such potential issues as when clocks become desynchronized and because the clock synchronization protocol opens another feature to attack [Gong92, Gong93]. However, Lam and Beth observed the timestamps were advantageous in a connectionless environment [Lam92], while Neuman and Stubbledine point out that timestamps avoid per connection state [Neuman93a].

Timestamps work by having a shared clock value between the clients and the drive. First, the drive checks that a request is within ε of the current time. If the request passes the initial check, the drive then checks that the request hasn't been seen within the window allowed by the first check. Passing both checks implies that the request has never been seen before and was recently sent to the drive.

NASD uses timestamp nonce based on clocks that are only synchronized in a very weak manner, thus avoiding many of the risks of systems with a clock synchronization protocol. All timestamps, and also access credential expiration times, are generated with respect to the drive's clock. The drive clock is *never* reset which would introduce security risks. Rather, the clock is a monotonically increasing counter that advances at a constant rate through time and the drive discards requests that were sent more than ε seconds from the current time.

For timestamps to be effective, the drive needs to keep track of the timestamps it has seen within a small window of the current time. The memory requirements correspond to the maximum number of requests the drive expects to see within the window. Deciding how much memory to allocate is essentially decides the maximum number of requests per second that a drive can service.

A filemanager or client can query the drive with any valid access credential to find out what the current time is at the drive. Since the time request has no side effects and releases no private information, the drive can respond to these requests even when the time request has a bad timestamp, which allows clients to bootstrap their relationship with a drive. This provides the filemanager and client with an idea of what the current time value is at the drive which they can then use in their requests. In the traditional sense, this is not a clock synchronization protocol because the protocol does not have consensus or agreement phase. The filemanager and clients are simply using the drive's clock as a counter; which advances at a roughly constant rate and can be readily reconciled with their own clock.

For each drive they are interacting with, the clients and filemanager must maintain a small amount of state: an offset and perhaps a scaling factor. However, the drive, the most resource-poor of the communicating parties, keeps no state other than its clock.

Before a client can communicate with a drive, the client must obtain an estimation of the drive's current time. As I already mentioned, the client can directly query the drive to get the drive’s current time. Alternately, when the filemanager sends the client its access credentials, the filemanager can also include the current estimated drive time, which reduces the number of message exchanges. Either way, the time value obtained must be trusted to be an accurate time value, in the sense that it is not forged rather, than in the sense that it is perfectly synchronized.

When a drive generates a reply, it increments the timestamp by one in order to allow the client to securely verify the relationship between the request and the reply. NASD timestamps are 64-bit values with theoretical nano-second resolution so it is extremely unlikely that a client will generate two requests with sequential timestamps and allow an adversary to replay the second as a reply to the first.

I selected timestamps in my design because they both fit more cleanly into the connectionless model of NASD and lend themselves to a simple way of reasoning about how much state to allocate to freshness. If NASD were to use a connection-based protocol rather than a connectionless, the fact that the sequence numbers likely already exist would make them more appealing.

4.5 Security Analysis

In this section of the dissertation, I discuss the result of applying a GNY analysis, described in Appendix A, and discuss denial of service in the NASD security system. While no security analysis is infallible, the combination of formal analysis with well-reasoned ad-hoc arguments presented throughout this chapter provides confidence that the basic NASD design is secure.

4.5.1 Discussion of Formal Analysis

Because of NASD’s unusual use of MACs, the relationship between public credentials and the keys, and the separation of the client-drive protocol from the filemanager-client protocol, it is hard to make any claim to having “proven” anything about the protocol. Nonetheless, the formal methods fulfill a very important role — they help to make explicit the assumptions that communicating parties are making and that I, as the designer, am making to allow them to reach their desired conclusions.

In Appendix A, I examine the client-drive protocol as well as a very simple filemanager drive-protocol to show how the drive is caused to believe a request came from an authorized client and how the client is made to believe the reply came from the drive.

The protocol must start with a few assumptions: the client and filemanager each believe they share a key, the filemanager and drive each believe they share a key, and the drive and clients believe that the filemanager can define shared keys between the clients and drives, and everyone can recognize a valid timestamp. Recognizing a valid timestamp is the freshness mechanism, which is described in Section 4.4.6. However, the details of how a client or drive verifies a timestamp are outside the scope of the GNY formalism so I must assume that the mechanism is correctly implemented by all parties.

In order for the formal analysis to complete, the drive needs to make a further assumption about how I generate private access credentials from the public access credentials, as described in Section 4.2.1. The drive must assume that the filemanager believes a MAC of a public credential is a valid key between the client with the rights described by the public credential and the drive. This new assumption is a formalization of the drive's belief in the validity of the MAC as a key generator. This assumption steps outside of anything that could be normally reasoned about in GNY, but is necessary to capture the core idea of private access credential generation.

With these assumptions, I show that the drive believes a request came from an authorized client and that the client believes the reply came from the drive.

4.5.1.1 Denial of Service

NASD is designed to resist state-allocation-based denial of service attacks similar to the TCP SYN attack [Schuba97]. A state-allocation denial of service attacks works by forcing the receiver to allocate some finite resource that is required for requests to be processed which prevents the resources from being used by valid requests. In the TCP case, a receiver is flooded with SYN packets, which forces the receiver to allocate resources on behalf of the sender. Since NASD is a connectionless system, our protocol doesn't require the drive to allocate any state in response to a request beyond the time required to reject or process the request. A NASD drive contains various caches which an attacker can potentially overflow. In order for an attacker to make requests that effect the data cache, key cache, etc., the request must first be validated before it is allowed to affect a cache. So, in order for an attacker to affect a cache, the attacker must be making a stream of unique valid requests which is much harder than merely generating a stream of simple packets.

NASD is subject to a denial of service attack against the drive's CPU or the drive's cryptographic resources. An attacker can flood a drive with bogus requests which forces the drive to expend computation to determine that the requests should be discarded. The drive can only discard a request after an attempt to verify the message authentication code has failed. If the drive can perform the necessary cryptographic verification at full line rate, then this is no more significant than simply consuming bandwidth. However, if the drive is cryptographically limited, the drive will spend time verifying the request. Unfortunately, there is no way to avoid the work of verifying the request without performing some verification so an adversary will always be able to execute some form of denial of service attack.

4.6 Implementation Status

The CMU NASD prototype has been up and running since 1997. The core drive prototype was written by Jim Zelenka, while I wrote all of the security-related code, although I integrated off-the-net 3DES and SHA-1 code from Bell Lab's CryptoLib library [Lacy93] and the Eric Young's SSleay libraries version 0.9.0b respectively. The prototype drive currently runs on both Digital Unix 3.2g and Linux as well as in the Digital Unix kernel.

Over the history of the prototype, I have adapted both NFS and AFS to run on top of NASD drives [Gibson97b]. Both prototypes use NASD security to enable the filemanagers to allow client access to storage devices. Additionally, AFS uses short-lived write access credentials to help enforce its quota management and to support its cache consistency model.

The prototype code was released in February 1999 to the Parallel Data Consortium and it was released to the general public in July 1999.

4.7 Related Work

4.7.1 Capability Systems

The NASD system passes around access credentials which behave very much like classical capabilities, first described by Dennis and Van Horn [Dennis66]. Capabilities are defined as "a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system" and are implemented as a datastructure that contains "a unique object identifier and access rights" [Levy84]. Historically, single processor or tightly-coupled multiprocessor capability systems have either used hardware support to prevent client modifications of capabilities or depended on trusted operating system kernels [Wilkes79, Wulf81, Levy84, Karger88]. In these systems, the capabilities were used as an access control resource because capabilities can be quickly tested for applicability to a given request.

In a distributed system, the untrusted network is introduced between communicating parties, so the problem becomes more complex. Some distributed systems such as Mach used capabilities to share resources and the capabilities were managed by mutually trusted operating system kernels [Sansom96]. The kernels managed the cryptographic keys necessary to send a capability to a remote machine. However, this security model fails when an adversary is able to modify the kernel on a workstation.

Amoeba proposed two solutions: trusted hardware and network enforced addressing [Tannenbaum86]. Using a mechanism that was based on one-way functions, Amoeba proposed special hardware called "F boxes" to manage capabilities over a sparse address space. The second alternative assumes that a client can never impersonate another client's address on the network. Thus, the network address is sufficient information to identify the origin. This assumption fails in many local networks such as Ethernet and in wide area IP networks.

Both Amoeba and Hydra [Wulf81] emphasized that capabilities provide a *mechanism* for implementing security and protection but give other parts of the system the latitude to choose the *policy* which is built on the capabilities. The goal of having NASD drives run under an arbitrary application made this quality of capabilities very appealing.

The ICAP system provides a capability model for distributed systems that is based on one-way functions [Gong89]. NASD is distinct from ICAP because of two critical differences: the use of MACs to generate keys and separation of issuer and verifier. In NASD, the access credentials include a private access credential, e.g. a cryptographic key, that is tied to the public access credential through a message authentication code rather than a one-way function. NASD access credentials provide a key to enable them to be used as a basis for encrypting or MACing an operations, rather than being a unique bit string as ICAP does. The second difference is who issues and verifies capabilities. In ICAP, the verifier of a capability must have access to an internal representation of the capability which includes a per capability secret created when the capability was issued. This implies that the issuer and verifier of capabilities are either the same entity or closely bound together. In NASD, the binding between the issuer and verifier is much looser and requires only a small set of keys to be shared between them. NASD achieves this by using a keyed function rather than an unkeyed function as the basis of access credential generation.

The scalability goal of NASD, which requires the server should not be involved in every request, motivated the high level communications flow which is very similar to Kerberos [Neuman94]. In Kerberos, the KDC grants clients kerberos tickets which enable a client to authenticate to an application server. In NASD, the filemanager grants clients a capability which enables a client to prove its rights to the NASD. Both Kerberos tickets and NASD tokens include a cryptographic key that the client uses in addition to some information that is securely communicated from the KDC or filemanager, via the client to the application server or drive. Another key difference is that Kerberos is transmitting encrypted keys to the application server while NASD is transmitting the information necessary for the drive to derive the cryptographic keys.

Neuman's proxy model is a similar model to the access credential mode of NASD [Neuman93b]. The emphasis of the proxy model has been on the power of the abstraction and integration with Kerberos, rather than an explicit emphasis on performance for a very narrowly defined application. Neuman also advocates that the mechanism be built on the authentication mechanism. Since NASD filesystems may use a variety of authentication systems based on a single drive interface, NASD requires the authentication mechanism to be separate from the base protocol to provide system builders the flexibility to choose their own authentication mechanism.

4.7.2 Other General Related Work

NASD provides the mechanism necessary to build a secure system which extends security all the way down to the I/O device level. However, higher levels of the system can choose to provide similar protection, in addition to or in place of NASD's security, by encrypting their data at the application level. A system built on the ideas of Matt Blaze's CFS [Blaze93] (such as the one proposed by StorageTek [Hughes98]) will protect privacy of data but will not protect privacy of communications. The distinction means that privacy integral to NASD will prevent different clients *who can all access an object* from having their requests read by their peers. Application-level privacy allows peers to read each other's requests because they all must access the data using the same cryptographic key. Another difficulty with filesystem-level privacy is the revocation of access rights. If the filesystem encrypts data before storing it on a drive and eventually needs to change the encryption key, such as when someone's access rights are revoked, then the filemanager must rewrite the entire file encrypted under a new key. For these reasons, NASD must provide privacy of communication for file systems that are not willing to sacrifice security, yet still want request privacy and efficient revocation semantics.

ISI's Netstation project proposes an alternative object-interface called Derived Virtual Devices (DVD) which attempts to solve the same problem as NASD (also discussed in Chapter 2) [vanMeter96]. When a drive boots up, it first authenticates to Kerberos [Neuman94] and then requests its basic configuration information from a remote controller called a Network Virtual Device Manager (NVDM). Similarly, the filemanager and client also authenticate themselves to Kerberos.

When a client wants to access data stored on a drive, the client makes a request to the filemanager, called STORM in the DVD system, for permission to access an object. STORM determines the best way to derive a virtual device for the client's operation. STORM then sends a message to the disk telling the disk to derive a new virtual device, i.e. defining the requested object as some aggregation of sub-ranges of an existing object — potentially the entire disk, and describing the rights being granted to the user. The drive then send an acknowledgment to STORM which, in turn, sends an acknowledgment to the client. This process has essentially shipped a filesystem object's metadata to the network disk. The client then uses its Kerberos tickets, which were probably obtained at the start of the session, and can make authenticated RPCs to the drive which matches the client's identity against the information provided with the virtual device was defined.

This exchange has two critical differences with NASD: added start-up latency and statefulness. The client can not begin requests to the drive for a minimum of two round-trip times plus queuing delays and service times on two machines. In contrast, the second message exchange is eliminated in the NASD architecture reducing the latency of the client's first operation. Secondly, when the filemanager defines a virtual device at the drive, the filemanager is consuming memory on the drive which limits the system's scalability as resources become scarce. While NASD requires the same overall system state, NASD maintains object metadata, equivalent to the definition of a derived device, on the storage media and does not associate it with active communications so the metadata can be discarded from memory.

Both NASD and DVD share a critical observation: security needs can differ from operation to operation and application to application. Both systems separate the data and the control portions of a request and allow them to be protected independently. This allows applications to trade-off the performance penalties of security in exchange for weaker protection in order to meet the its needs.

IBM's SCARED system is another project addressing the same problem as NASD along with the security problems [Reed99]. Reed et al. are extending the NASD abstraction but subtly differ on many details. SCARED rejects placing privacy in the storage interface but rather, believes that the application level can provide all privacy services. Earlier in this section, I argued why this approach can be inappropriate for some applications.

While NASD uses a message authentication code to derive keys, SCARED uses a cryptographic hash function of the key append to the public rights description. Both are strong ways of establishing the relationship between a client's rights and their cryptographic keys. A strong MAC, such as HMAC-SHA1 which is used in NASD, will prevent an adversary from prepending information to the rights description. In theory, an adversary can prepend data to the public rights description in the SCARED solution. Since the initial vector of the hash is publicly known, the adversary may find a data block such that the hashing the data block results in the hash function having the same internal state as its publicly known IV. This data block could be safely prepended to any public rights description without detection.

SCARED also places more functionality at the drive than the NASD system. In addition to using capability keys like NASD, it also includes identity keys which requires the drive to make an access control decision. This places policy decisions at the storage device rather than in the filemanager. When a client makes a request to the storage device, the client can use either an identity key or a capability key to make a request from storage. However, the storage device always generates a reply using an identity key, derived in a similar manner to client identity and capability keys, so that the client can verify the origin of the reply.

Because the drive uses a client-specific identity key on a reply, a SCARED system can publish the access keys for a group of objects, eliminating the need for a client to obtain them from the filemanager, without allowing clients to impersonate the drive. However, each client must obtain a unique identity key to verify replies from the drive. This features enables SCARED to grant a group of users access to a group of objects while each user only requests a single key from the filemanager. In Chapter 5, I present several aggregation mechanisms that can achieve the same result within NASD.

Additionally, SCARED places the directory management function into the storage device rather than in a filemanager machine. At a high level, these illustrate the difference in the amount of complexity that SCARED puts in the storage device in comparison to the NASD architecture.

NASD assumes that clients and drives all have clocks that advance at a predictable rate, which are minimally necessary to limit the lifetime of access credentials, and can also be used to check the freshness of operations. SCARED allows the same mechanism, called timers in the SCARED design, and also includes a stateful sequence number approach. SCARED is defined to handle both stateless RPC style semantics as well as connection-based semantics which NASD does not address. The NASD security system could also use a sequence number system at the cost of latency to initialize sequence number state. If the drive were also free to discard connection state, then the client would suffer additional latency to resynchronize sequence numbers when resynchronization was necessary. NASD adopted a timestamp based design because it emphasizes both low latency and low state overhead.

The Echo filesystem developed at Digital’s System Research Center implements a model very similar to NASD and uses the same mechanism to manage cache coherency [Mann94]. A client’s permissions are stored in an ACL that is checked by the server on an initial access. The client’s rights are then wrapped up into a token which the client can use for future accesses that bypass permissions checking and maintain cache coherency through the same mechanism. Echo’s designers adopt this policy, in part, because ACL checking is more time consuming than simply checking that a token is valid for an operation. Unlike a NASD system, the servers must maintain a list of all outstanding tokens in order to properly enforce their coherency protocol. Within NASD, a similar effect can be achieved with short-lived access credentials and temporarily revoking access credentials by modifying the access control version number and restoring the access control version number when the synchronized operation completes. While the access control mechanism is a tool that an application can use to help enforce cache coherency, the application is ultimately responsible for managing consistency using NASD’s mechanisms to control access to storage.

4.8 Summary

In this chapter, I have presented a basic design for a security system for network attached storage which has been implemented in CMU’s NASD prototype system. The design defines how a filemanager grants access to a network storage device but does not specify the policy with respect to rights are granted. By leaving the policy decision in the filemanager, the design allows a variety of access control policies to be implemented on top of the basic security system.

The design minimizes the impact of security on the overall system by maintaining RPC semantics, as opposed to a connection-based system, and asynchronously involving the filemanager. Since drives will continue to have very limited amounts of RAM, the stateless RPC protocol is most appropriate. The design requires the filemanager to be involved on an initial access to a NASD object but further accesses bypass the filemanager while still synchronously enforcing the filemanager's policy decisions.

A filemanager enables clients to access storage by securely giving a client a token, called an access credential, which includes a description of access rights (the public credential) and a cryptographic key (the private credential). The client uses the access credential to prove to the storage device that the client is entitled to perform a specified operation. The cryptographic key is used to both cryptographically bind a request to the description of the client's access rights (using a message authentication code), as well as to protect the integrity of a request and reply (using a message authentication code). The key can also be used to protect the privacy of requests and replies.

The security of the system is based on a few basic assumptions:

- A small number of cryptographic keys are privately shared between the filemanager and drive.
- It is infeasible to forge a message authentication code (HMAC-SHA1).
- The message authentication code is a good keyed pseudo-random number generator that can be used to generate cryptographic keys.
- The cryptographic hash function (SHA1) is one-way.
- The encryption algorithm (3DES) is resistant to attack.

My design and implementation specify certain cryptographic functions. The same basic design could be implemented with different encryption functions, message authentication codes, or cryptographic hash functions, although the security and performance would be different.

While constructively describing the design of NASD, I have argued various points of security which are summarized in Table 4-1. A secure system can never be proven to be entirely secure, but the table summarizes why the NASD security system is resistant to attacks.

Table 4-1 Attack-Countermeasure Summary

Attack	Countermeasure
master, drive, or partition key recovery	•use limited to key change operations and infrequent allocation changes
working key recovery	•difficulty of key recovery attacks against HMAC-SHA1 •key lifetime contains available text-MAC pairs •key length
forged private credential	•difficult of key recovery attack against HMAC-SHA1 •difficulty of forgery of HMAC-SHA1 •working key limits available text-MAC pairs •key length (160 bits) •working keys only known by filemanager and drive
forged request	•forgery resistance of HMAC-SHA1 •access credential lifetime limits available text-MAC pairs •partition *must* require integrity protection •credential lifetime constrains available text-MAC pairs •privacy protection can further reduce available data to attacker
data release	•randomness of SHA-1 to generate the IV •strength of Triple-DES •strength of counter mode •partition *must* use privacy protection options
delayed request	•timestamp present in a request •drive-centric timestamp; drive’s clock is monotonic •partition *must* use integrity protection
replayed request	•timestamp in a request •drive-centric timestamp; drive’s clock is monotonic •ε window of acceptance •partition *must* use integrity protection
replayed reply	•function of request timestamp included in the reply
force encrypted data to be written as plaintext	•partition *must* use integrity protection •ordering of encryption and MACing
state-based denial of service	•stateless semantics make it hard for a adversary to consume state
cpu-based denial of service	•none unless cryptographic hardware can perform at line rates

Chapter 5: Alternative Access Credentials

A capability, the basic type of NASD access credential described in Chapter 4, is a natural mechanism for the filemanager to package up its access control decisions and communicate them asynchronously to a drive. The filemanager is responsible for all the application-specific complexity, such as processing ACLs in NT or accessing NIS group databases in NFS, while the drive performs a lightweight access check, simply verifying that the capability is applicable for the requested operation and that the capability has not been revoked. In the common case of access control policies which seldom change, the complex portion of an access control decision is performed once by the filemanager and the drive performs a much simpler verification task on a per request basis. The designers of DEC SRC's Echo filesystem performed a similar optimization to avoid the complexity of a full access check on a per request basis by combining their access control and cache consistency into a token-based system [Mann94].

Unfortunately, capabilities have several disadvantages. They are tied to a single NASD object which requires the client to obtain at least one capability per object accessed. Requesting capabilities can add both latency to the client and account for a substantial portion of the filemanager's load. All the burden of the access control decision is placed on the filemanager, so when a change in access control potentially affects a large number of objects, the filemanager must explicitly revoke all outstanding capabilities, either by touching each object's metadata or changing a high level key, in order to enforce the change. Additionally, capabilities are a poor fit for operations on multiple objects because the number of keys and size of the access control information included in each request will increase linearly with the number of objects operated on.

In this chapter, I first explain the shortcomings of the NASD capabilities, presented in Chapter 4, and then explore several alternatives, each an improvement over its predecessor. One of the motivations for the alternatives is the load that a capability system places on the filemanager system. I quantify this load through a trace-driven simulation study. The first alternative is to modify capabilities to explicitly name multiple objects rather than a single object in the public portion of the capability. The second alternative introduces a layer of indirection in naming of objects through group objects that store their membership lists on-disk. Finally, I present a solution using small executable access credentials which can read a requested object's metadata and return an approval or failure code which I have implemented in a prototype NASD.

5.1 Critique of NASD Capabilities

The original NASD capability mechanism was closely tied to the NASD interface, which allowed me to optimize for drive complexity and size of the capabilities. As a result, the drive's role in checking access permission is minimized to a few simple equality checks. Capabilities were also small — a 48 byte public portion plus the 20 byte private portion — because they explicitly name only a single object. The small size minimizes the amount of calculation necessary to generate the private credential of the capability from the public credential, as well as limits the overhead of moving the capabilities on each request, storing the capabilities, and the amount of space consumed in on-drive access credential caches.

The capability model requires no potentially expensive I/O operations beyond the operations required to service the same operation when the drive provides no security. In a modern Seagate ST31903 series drive, average seeks are 5 msecs [Seagate99a], which is a significant portion of a request's service time in modern systems where media transfer rates and network interface rates are both in the 100 Mb to 1 Gb range. To enable low latency servicing of requests, it is important to minimize the number of different I/O operations necessary to service a request. Capabilities achieve this goal by using only a small amount of information to decide if a capability is valid: the client's request, the capability included with the request, and the access control version number which is stored in the requested object's metadata. The requested object's metadata must be retrieved, since the CMU prototype uses a Berkeley FFS-like inode structure [McKusick84] to locate the object's data blocks. Thus, decisions are *local*: that is, they do not require any data beyond the access credential, the request, and the requested object.

Next, I will discuss the following problems with the basic capability model: no support for dynamic relationships, client latency for credential requests, the load offered to the filemanager, lack of support for operations involving multiple objects, and no cross-object locality for keys. These problems will be used in later sections to motivate improvements on capabilities.

5.1.1 No Support for Dynamic Relationships

A capability system can not efficiently implement revocation in an application that uses dynamic inheritance of access control permissions. In some filesystems, including Novell [Sheldon96], Appleshare [Apple98], and Windows NT [Frisch98] (with bypass traverse checking disabled), files and directories dynamically inherit permissions from their ancestors at request time rather than at create time, as discussed in Chapter 3. In dynamic inheritance systems, if permissions are changed on a high-level directory, then the change will immediately affect the files and directories beneath it. With capabilities, the only way to enforce this change is to revoke all capabilities potentially outstanding on the changed object's descendents, which involves either changing both working keys to render all outstanding capabilities invalid, or changing the access control version on each of its descendents. Both are expensive operations.

Introducing a dynamic relationship between objects could also allow more timely revocation when group memberships change. Most of network file systems I investigated failed to enforce changes in group membership on the next request after the change occurred. Instead, the filesystems take a lazy approach and enforce group membership changes when the user next authenticates to the system or when a client starts a new connection to a server. The likely reason for this is to avoid an expensive query to a group database server such as the PTS server in AFS or the domain controller in NT. Other filesystems may take a more aggressive stance and attempt to enforce group changes when they occur rather than deferring them until later like the systems I surveyed. Efficiently implementing quick group changes poses a problem similar to dynamic inheritance because quick group changes introduce a dependency between access control decisions for all objects that reference a group and the current membership list of the group.

These two examples suggest that some mechanism for capturing a simple but dynamically evaluated relationship between NASD objects would be useful to some applications.

5.1.2 Client Latency

When a client accesses a series of objects, it may make numerous requests to the filemanager each which have a latency penalty. These requests occur because a capability explicitly names a single object for which it is valid so a client must issue an RPC to the filemanager every time the client touches a new object. Figure 5-1 shows an example of the messaging behavior that the client would generate when she is examining the attributes of all files in a directory or recursively searching through a directory tree. If the drive is able to successfully prefetch a future access, based on the **nearby object** attribute in the NASD interface [Gibson97b] (which is similar to deJonge's logical disks [deJonge93]), then the expected service time for a request, in the absence of security, will be small and the message traffic to request capabilities will be a significant portion of the overall request time.

5.1.3 Load Offered to Filemanager

Clients requesting capabilities from the filemanager create a significant portion of the workload offered to a filemanager, affecting scalability. Every time a client requests a capability, the filemanager performs its application-specific procedure to determine what rights to grant the client. This is exactly the part of access control that the Echo filesystem attempted to avoid because it was expensive. Independent of the application-specific costs, the filemanager must service an RPC from the client and generate the private portion of the access credential (i.e. generate a MAC of the public access credential).

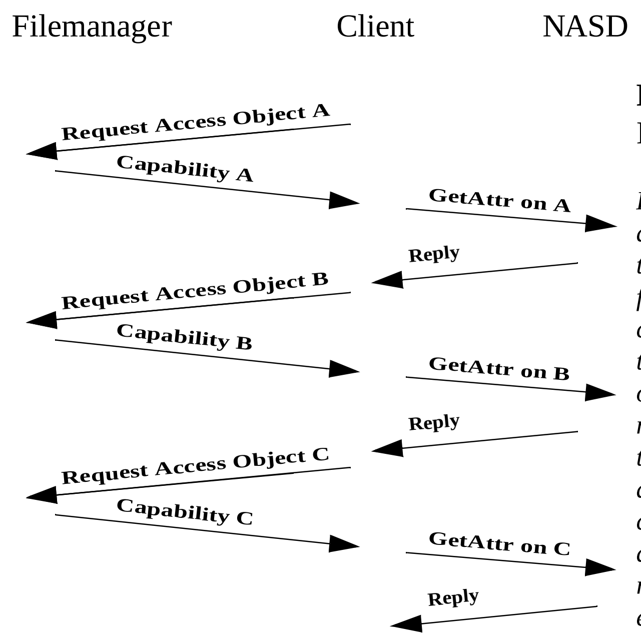


Figure 5-1 Capability Requests Impact on Latency

For a sequence of small requests, such as a recursive find on a directory tree, the additional round trip RPC to the filemanager to obtain an access credential can add substantial latency to the client's operations. For each object the client touches, the client must obtain a unique capability from the filemanager. This graph illustrates a request pattern where this request can double the request latency if a drive successfully prefetches the requested object's attributes, thereby eliminating media time.

Earlier PDL research [Gibson97a] showed that the NASD architecture can reduce fileserver, or filemanager load (during burst activity) by a factor of up to five for AFS and up to ten for NFS. Capability traffic is a substantial portion of the remaining load. By moving all the data movement tasks off of the filemanager, the filemanager's primary task becomes maintaining the integrity of the application's structures, such as the namespace, and enforcing security policies. Analytical modeling showed that during the busiest 2% of minutes in the NFS workload, capability traffic would account for 73% of the filemanager requests. Similarly, during the busiest 5% of the minutes in the AFS workload, the capability could account of 79% of the requests. Thus, the load generated by capability requests is a significant portion of the filemanager's load.

In the remainder of the section, I address two key question via simulation. First, how much of the request traffic to the filemanager is a direct result of capability requests? A higher offered load implies filemanager is performing more work and thus requires more resources to service clients. Secondly, how much does the maximum load with capability requests differ from the maximum without capability requests? If the maximum load for a given percentage of the active minutes with capability traffic is higher than maximum load for the same percentage without capability traffic, then the server will need more resources to promptly service that percentage of the client minutes if the managers wish to minimize customer dissatisfaction [Riedel96].

5.1.3.1 Sample Workloads

I studied two workloads: a Berkeley NFS trace and CMU AFS 1999 traces. The Berkeley NFS trace records activity on an Auspex fileserver supporting 231 client machines at the University of California at Berkeley for a one week period [Dahlin94]. Berkeley researchers post-processed their trace using heuristics to eliminate attribute reads that were involved in cache consistency, so the trace is an approximation of the actual workload. Because this change increased the load on the filemanager and reduces the impact of capability traffic on the load, I chose to continue to use these traces which are already familiar to the research community. The NFS trace is dominated by overnight backup activity. Since users are mostly insensitive to backup performance, I exclude these periods of the trace and only studied requests timestamped Monday through Friday between 9am and 5pm. This is the same portion of the Auspex trace that was used in earlier NASD research [Gibson97a].

For comparable AFS traces, I used traces of traffic to the Parallel Data Lab’s AFS server over two Monday to Friday time periods: January 25th-January 29th (AFS Week 1) and February 8th through February 12th (AFS Week 2). The server primarily holds archival and working directories for a research group of 30 people involved in research and software development in an academic environment. The server was accessible to anyone connected to AFS but the primary consumers of the server’s resources were members of the research group. Unlike NFS, AFS backup traffic bypasses the server so I was able to use data for all 120 hours of each 5 day trace.

Both workloads mark each log record with a timestamp, unique client-host ID, and an indication of the request type.The NFS trace only records the general class of issued request which leaves some ambiguity in determining exactly which primitive NFS requests were issued (e.g. a request recorded as a directory read may have been either a lookup or a readdir request). The AFS trace records precisely specify the request type, including a unique client userid, and identity of the parent directory in which the requested object resides. This allows a more accurate modeling of filemanager load because multiple requests from different users on a single client host are not aggregated behind the client host ID. The presence of the parent directory allows me to recognize when multiple filesystem objects are in the same directory, which will be used in simulations presented later in this chapter.

5.1.3.2 Simulation

For both workloads, I simulated a NASD version of the filesystem’s behavior where clients generated their own capabilities versus one where the filemanager was responsible for generating capabilities. When clients generate capabilities, there is very little security in the system because a large number of clients is more likely to be compromised than a few well-maintained servers. In both case, the filemanagers are still responsible for maintaining the integrity of the filesystem structures, so filemanagers handle operations that create or destroy files/directories or modify metadata. In the AFS simulator, the filemanager is also responsible for callback management to provide advisory cache consistency. The difference between client- and filemanager-generated capabilities is in how a client goes about getting a capability to read or write file data. Clients generating capabilities can read arbitrary files or metadata and write files but are unable to destroy the structure of the filesystem because the integrity critical operations still go through the filemanager. Client-generated capabilities are useful as a strawman to understand the filemanager's cost of handling requests for capabilities.

Optimistically, I assume that clients will hold onto capabilities and reuse them until they expire. This is similar to how users manage AFS tickets when they are logged onto a workstation for a long period of time [Satyanarayanan89] and generates the fewest capability requests for a workload. An alternative mode of operation would be for clients to discard capabilities when a file is closed, or shortly thereafter if they are not reused, much like operating systems cache the mapping from a file name to a filesystem-specific file handle. This approach requires clients to requesting capabilities more often thus increasing the filemanager's load.

For each trace, I simulated clients to find when they would request a new capability from the filemanager. Simply understanding the offered load to the filemanager captures the first order effects of how capability request traffic will impact the filemanager and can serve as useful barometer to compare design options. A client requested a new capability when it needed to perform an operation (such as a read) that would go directly to storage, rather than being handled by the filemanager (such as file creation), and the client did not currently hold an appropriate capability. For both AFS and NFS, clients were issued capabilities valid for twenty-four hours which is equivalent to the default lifetime of CMU's Kerberos tickets. The NASD on AFS filesystem also utilizes short-lived, (20 seconds in my simulation) capabilities to enforce a write-lock as well as quota enforcement despite the filemanager not being on the datapath for the write operation [Gibson97b]. With both client- and filemanager-generated capabilities, the client informs the filemanager when it is writing a file so the filemanager knows to break outstanding callbacks. In contrast, NFS clients are given single capabilities with their complete access rights for the specified object.

For the NFS simulation, I simulated a system where the clients directly parse directories, which is already done by clients in AFS. Without this optimization, which was discussed in [Gibson97a], much of the scalability advantage of NASD is lost and the offered load increases by a factor of 23 without accounting for capability requests, due to client requests to the filemanager to parse file names (i.e. the Unix lookup() call).

In the AFS simulation, I assume that clients batch together requests to the filemanager when doing a **BulkStatus** operation. In a **BulkStatus** operation, the client requests the attributes of up to 20 different files. When communicating with a NASD drive, the client will break this into up to 20 different **GetAttribute** RPCs because of the limitations of the NASD interface. But the AFS protocol already uses a single RPC for a BulkStatus so it is natural for AFS on NASD to perform a single request to the filemanager to request the necessary capabilities.

For all three workloads, the capability traffic substantially increased the number of requests that clients make to the filemanager. The overall offered load increased by a factor of 1.8, 1.4, and 2.8 for the AFS Week 1, AFS Week 2, and NFS trace respectively.

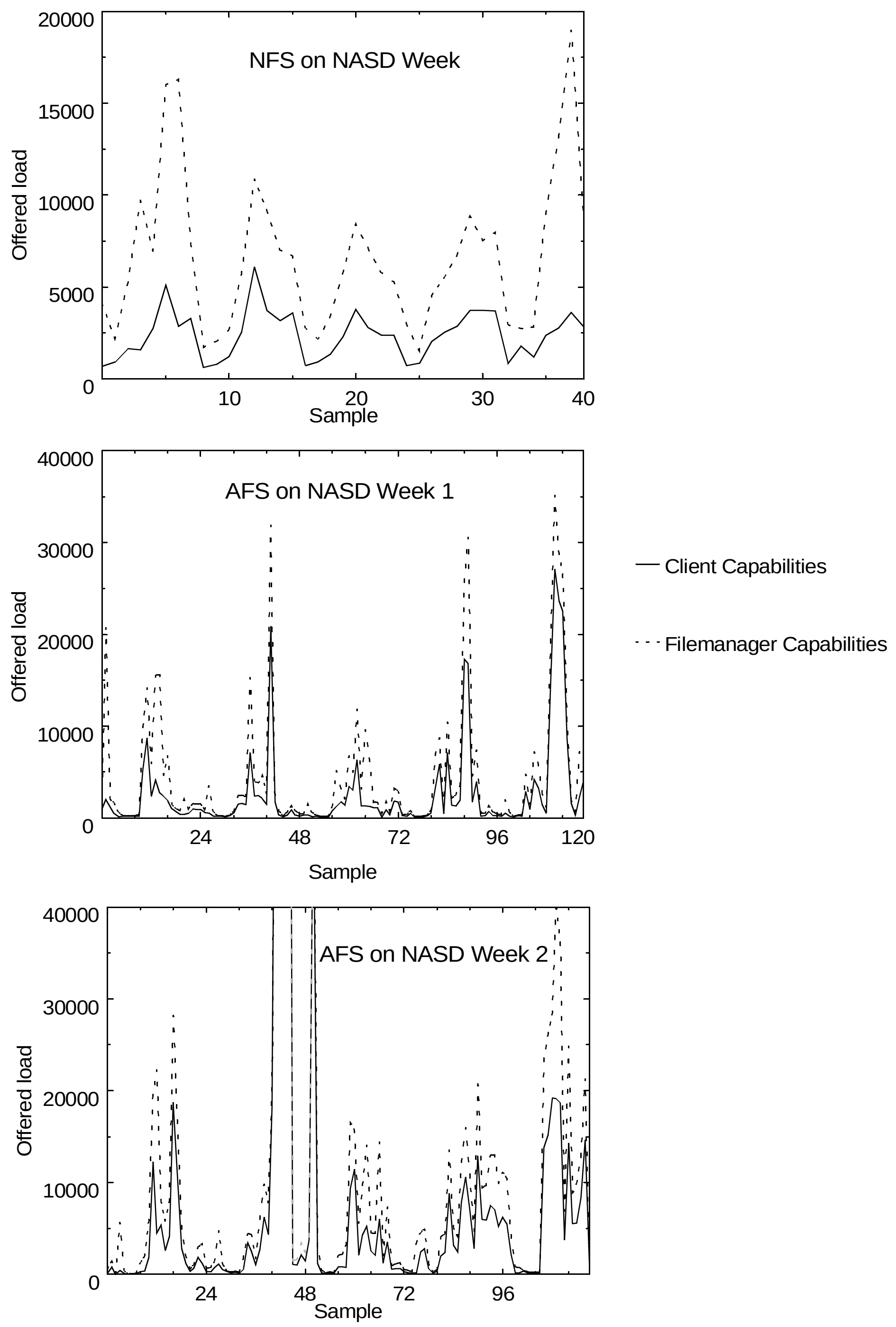


Figure 5-2 Offered Load Over Entire Trace

These graphs show that particularly for NFS, the task of generating access credentials significantly increases the filemanager's load. AFS presents more data on the same-sized graph, so the effect is less obvious. Each graph plots the simulated offered load to the filemanager sampled over 1-hour intervals in two cases: the filemanager generates all capabilities or clients generate their own capabilities. At some points of the graph, lines appear to disappear because they are essentially equal to another line which obscures it.

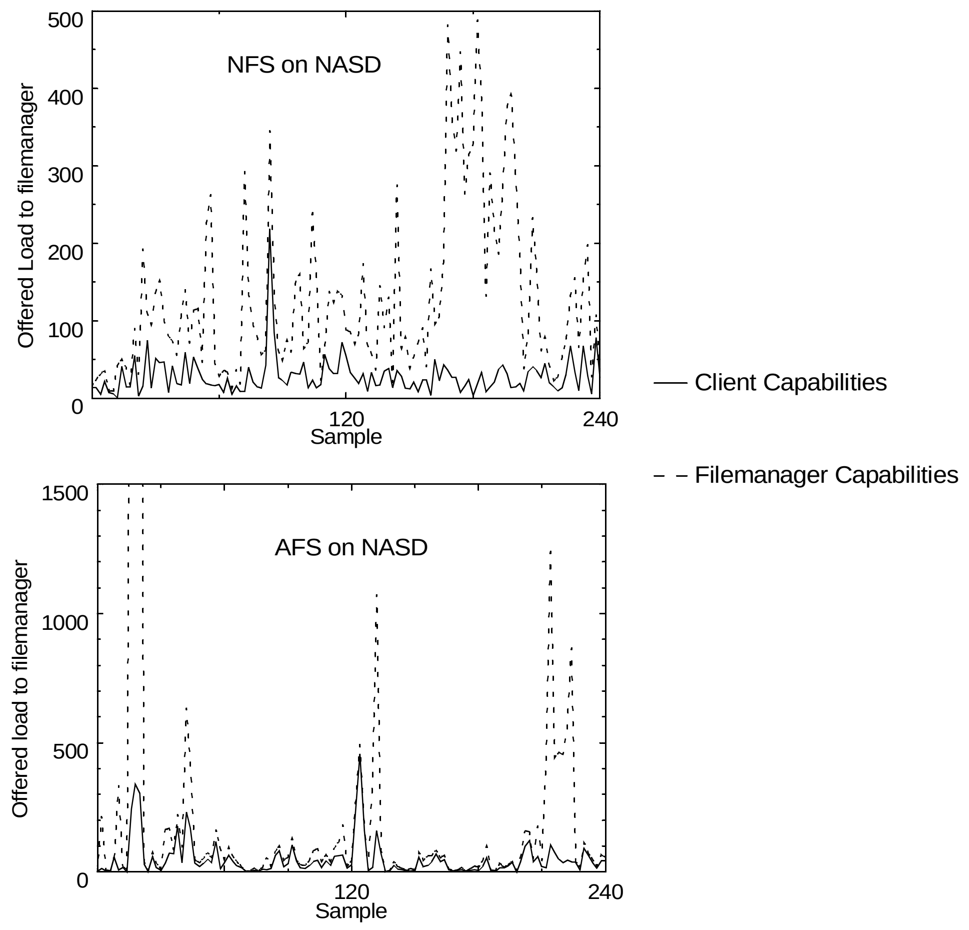
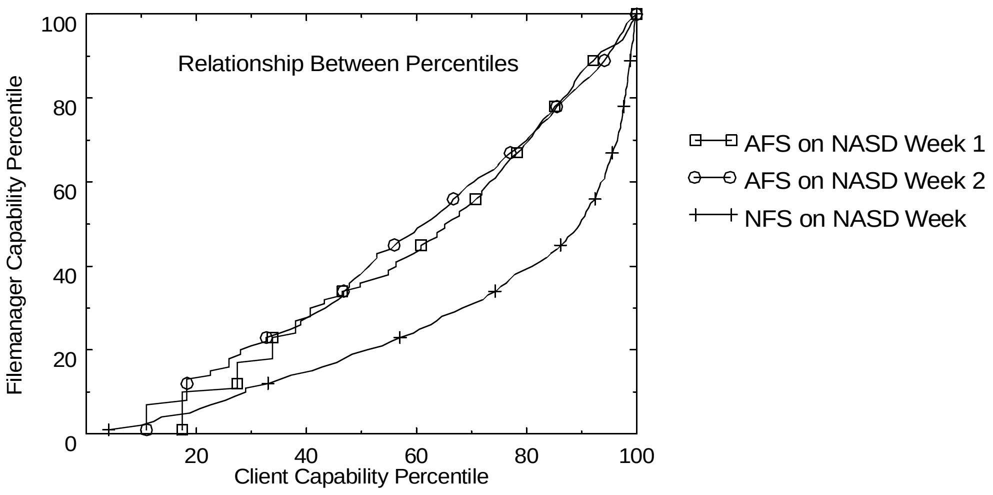


Figure 5-3 120 Minutes from AFS and NFS Workloads

Both graphs show that a large amount of burstiness, which can cause noticeable delays when the bursts go above the saturation threshold of a filemanager, can be attributed to a filemanager generating capabilities. The y-axis illustrates load offered to the filemanager over 120 minutes measured in 1-minute intervals. for both types of workloads. Each of these graphs corresponds to 2 samples in the Figure 5-2.

This point is illustrated in Figure 5-2, which shows each trace in 1-hour long samples. The figure also shows how the capability requests can significantly amplify the size of the request bursts. At a finer time-scale, Figure 5-3 shows that capability traffic significantly increases the amount of variation in the server load, which will require a more powerful server machine in order to minimize the impact of the bursts on clients.

Figure 5-4 shows how provisioning filemanager resources for some level of performance when not generating capabilities will correspond to the filemanager's ability to handle the same workload when filemanagers are generating capabilities. For both AFS traces, a server designed to service requests up to the load of 80% of the minutes would only be able to handle the load in 70% of the minutes if the server needed to generate capabilities. In NFS, the difference is more pronounced: a filemanager designed for 80% of the minutes without generating capabilities would only be sufficient for 31% of the



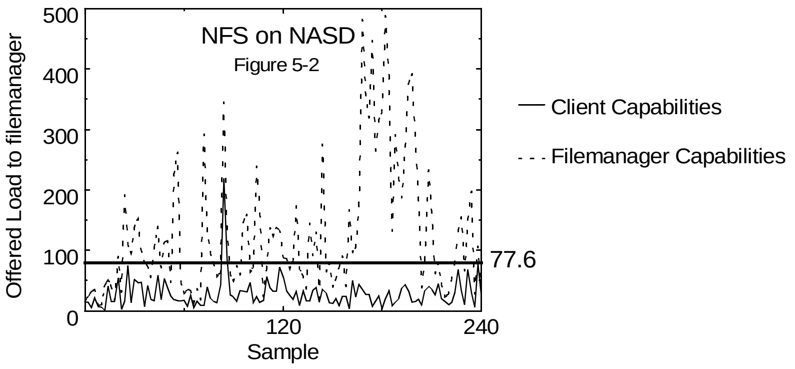


Figure 5-4 Filemanager Load Percentiles: Capabilities

This graph shows the relationship between the percentage of minutes below a threshold for both client and filemanager generated capabilities. The y-values are the percentage of active client minutes with filemanager generated capabilities that are less than or equal to the maximum of the x-value's percentage of client generated capability minutes. For example, the NFS on NASD line shows that 90% of the client generated capability minutes have a load less than or equal to the load of the 51% filemanager generated capability minutes. In the lower graph (a small version of the NFS portion of Figure 5-2), the horizontal line shows the 90% mark for client generated capabilities which is a load of 77 requests. Only 51% of the filemanager generated capability minutes lie below this same line. The curve will always be at least slightly convex because the offered load to the filemanager when it generates capabilities is always greater or equal to the offered load when clients generate their own capabilities. A linear relationship would indicate that the offered loads are with filemanager generated capabilities and client generated capabilities were equivalent.

minutes when called upon to generate capabilities. This difference is a result of NFS providing less functionality (specifically relatively weak consistency guarantees), thus the filemanager performs less work overall in the NASD architecutre. As a result, the task of an NFS on NASD filemanager is much smaller, primarily to protect filesystem integrity, than AFS on NASD which implies that both the scalability gains from NASD are greater for NFS [Gibson97a] and the impact of capability traffic is larger.

The simulations have shown that capability request traffic can increase the load offered to the filemanager by a factor of 1.3 to 2.8, as well as amplifying the burstiness of the workload. A filemanager that needs to handle capability requests in addition to its other tasks requires greater resources than a filemanager whose clients can independently generate capabilities. However, a system where clients can generate capabilities is very insecure because there are so many weakly protected points of failure and the filemanager must generate capabilities in order to construct a secure system. In order to reduce the impact of capability requests on the filemanager load, I will present alternatives to capabilities that both reduce the offered load to the filemanager and smooth out the peaks created by access credential traffic, the general class of tokens of which capabilities are the simplest example.

5.1.4 Operations Involving Multiple Objects

Many filesystems could take advantage of a mechanism to perform operations on multiple objects with a single request. For example, both AFS's BulkStatus [Transarc91] and NFSv3's ReadDirPlus [Callaghan95] allow clients to retrieve status information of multiple objects with a single request to the server. NT has directory scanning RPCs, the FIND_* calls, and an operation chaining mechanism, the ANDX mechanism, that can provide equivalent functionality as well as enable sequences of operations on a single object with a single RPC [Leach97b]. In the initial NASD interface [Gibson97b], there are no operations that operate on multiple objects and the capability model has discouraged efforts to add any such operations. These types of operations are desirable because they reduce client latency and offer more work to a drive in a single operation, which allows the drive to make more efficient scheduling decisions about its resources.

A set of capabilities could be used for a multi-object operation but this requires the client to send all the capabilities to the drive. Additionally, some well-defined combination of their private credentials must be combined to generate an access key to authorize the request. And finally, the drive needs to individually verify that each capability is currently valid. A more elegant solution would involve a single access credential that enabled the entire operation which avoids some of the overhead of processing and handling multiple capabilities.

With multiple capabilities being used for an operation, all the access keys are combined into a single key and if one of the capabilities is invalidated, by updating its access control version number (discussed in Chapter 4), then the entire operation will fail because the combined access key will become invalid. If a single more general access

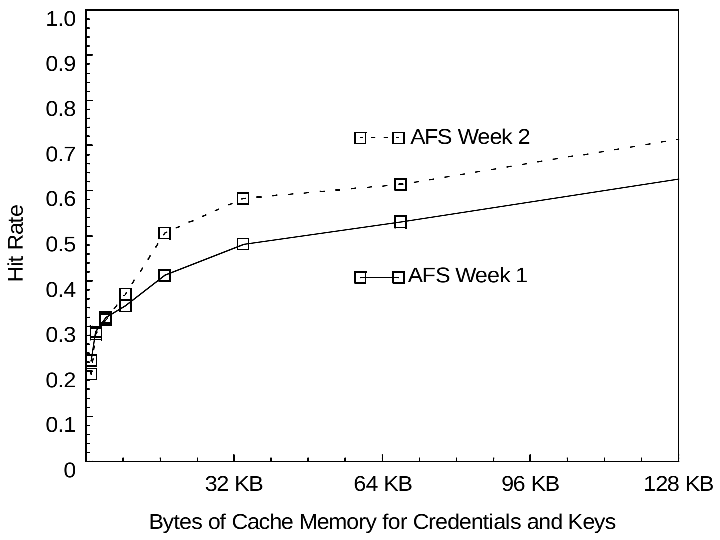


Figure 5-5 Baseline Hit Rate in On Disk Capability Cache Performance

Caching access credentials only hits 70% of the time in an on drive credential cache. This figure shows the hit rate using two AFS workloads to simulate a cache on a NASD drive that maps the public portion of an access credential to the MAC and encryption keys. Clients are given unique short-lived capabilities for write operations in addition to their read access credentials. Clients are given capabilities that enable read and getattr access to a single object. Each cache entry contains two 20-byte keys per cache entry and 48 bytes of public capability data.

credential is used, I can separate out the key generation from the metadata of the object and the access key can be generated regardless of any given object's access credentials.

5.1.5 No Cross-object Locality

Capabilities in an on-disk access credential cache, described in Section 4.3.2.2, capture locality only among references to a single object. Other common abstractions such as a directory, ACL, or user ID can capture more locality. By more locality, I mean that a drive that associates an access credential with a user accessing a specific directory will see greater reuse of the access credential than a drive that associates an access credential with a single object. Capturing this additional locality will improve performance in drives that are sensitive to the cost of computing an access credential's cryptographic keys.

Some NASD drives will lack hardware support for cryptography which makes the recomputation of keys necessary, giving a sizeable performance penalty, when an access credential is not found in the on-disk credential cache. Performing the cryptography to generate the key for MACing requests requires about 6000 instructions in CMU's prototype running on an Alpha 21064 processor. For a small request from a client, such as an 8K cold cache read which normally requires 67K instructions in the prototype [Gibson98], computing the proper MAC key increases the number of instructions executed by 9%. This is an optimistic measure of the cost of generating the MAC key because the prototype uses DEC’s implementation of DCE, a heavyweight RPC mechanism, which accounts for 79% of the cycles. When the drive uses a lighter-weight and more optimized RPC layer, the base instruction count for an operation should become smaller and the penalty of extra cryptographic work will increase. For example, if I assume that a lighter-weight RPC uses a factor of 4 fewer instructions, reducing the percentage of instructions spent in communications from 79% to 49%, and an SHA-1 implementation optimized for a factor of 2 reduction in instruction, then the penalty for the additional cryptography increases to 11%.

If the drive designers choose to implement a capability cache, a simple LRU based caching scheme will deliver the hit rates shown in Figure 5-5. Since capabilities are only valid for access to a single object, the cached entries capture a limited amount of the locality in the workload and the knee of the hit rate curve is quickly hit with only 32K of memory with a 60% hit rate. As a result, the drive will need to compute the cryptographic keys for a request for 40% of the requests which, as discussed above, may result in a significant performance cost.

5.2 Batching Capabilities

Clients can reduce their latency by consulting the filemanager less frequently to obtain capabilities. In order to achieve this goal, clients must obtain more capabilities every time they contact the filemanager. A client could batch a group of capability requests together to the filemanager in a single request to the filemanager. The AFS simulation in Section 5.1.3 batched together client capability requests necessary for a single BulkStatus() operation. Alternately, the filemanager could predict future client accesses based on past behavior and prefetch capabilities using techniques similar to predictive data prefetching [Griffioen94]. However, clients generally have more information about their access patterns than the filemanager so they are better prepared to prefetch capabilities via a batch capability request mechanism than a filemanager speculating about client patterns. Simply batching or prefetching groups of *N* capabilities requires the client to contact the filemanager less often but still requires the filemanager to generate *N* distinct capabilities to return to the client. Filemanager load will be reduced because communications overheads of the client-filemanager RPC will be amortized across multiple capabilities, but the per-object access costs associated with *N* access checks and computing *N* access keys remains. Perhaps more importantly, the client machine must have enough information from the user to request the capabilities in advance. If the application interface does not pass sufficient information from the user to lower levels of the system, the filemanager and the application’s client-stub will be unable to usefully predict the user’s future accesses. Transparent informed prefetching [Patterson95] could be used to augment any application to transfer information about future object accesses from the user-level to the application level and allow capabilities needed in the future to be prefetched in batched in requests to the filemanager.

By combining the replies together in a single result, which I call a *list capability*, a filemanager can reduce the cryptographic costs of generating the capabilities. The fixed cryptographic costs of generating a capability are amortized across multiple objects rather than being multiplied by the number of objects a client is enabled to access. Extending the definition of a capability given in Section 4.2.2, I replace both the Object ID and AV in the capability with a list of (NASD ID, AV) pairs and generate private credential as follows:

$$PrivateCred = MAC_{K_x}(PublicCred, ListLength, ListOf(ObjectID, AV))$$

Clients now receive a single large capability that enables access to a number of different NASD objects. Because objects are explicitly named in the list capability, its size is directly proportional to the number of objects for which it is valid. The disadvantage of size are both that the list capabilities have a bigger footprint in drive access credential caches and a large list capability requires more work to generate an access key. Also, access credentials are sent on every request, because of the statelessness of the NASD protocol, so larger access credentials will have a larger communications overhead. Finally, the list capabilities do not address the need for capturing dependencies between objects using the access credentials.

5.3 Indirection via On-disk Objects

A further enhancement is the introduction of a level of indirection into the naming of objects accessible with an access credential by allowing the access credentials to not explicitly name all the applicable objects. Concretely, a filemanager gives clients either a normal capability or a *group capability* which names another object, called the *indirection object*, which in turn contains an ordered list of object identifiers for objects in the same partition. The client can use the group capability to perform operations on any object listed within the indirection object. The access key for a group capability is generated from the public credential, describing the bearer's rights, and the group object's ID and AV:

$$PrivateCred = MAC_{K_x}(PublicCredential, GroupID, GroupAV)$$

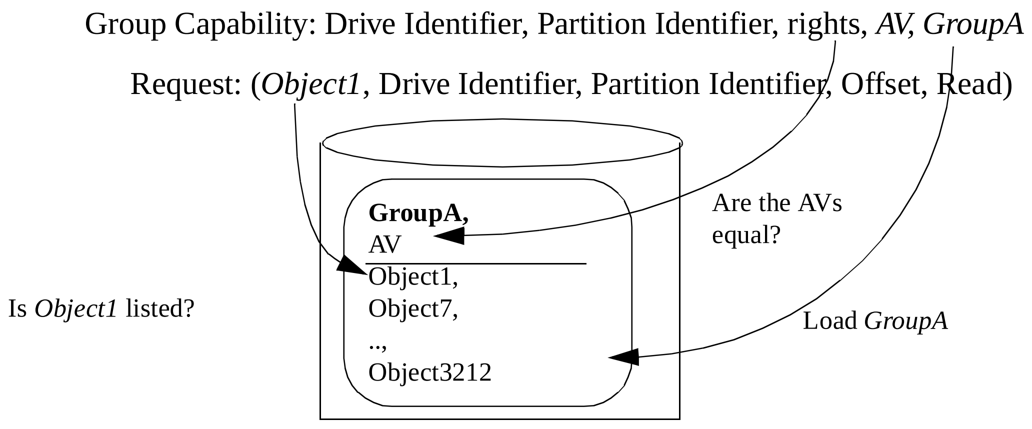
A filemanager can revoke access to an entire group or a single object through the group. Every indirection group has its own AV which the filemanager can use for wholesale revocation of all access rights enabled through the indirection group. Access to a specific object through the group can be enabled or disabled by adding or deleting the object from the indirection group without affecting access to any other object referenced in the indirection object.

Groups are a common abstraction, frequently used to improve efficiency or manageability, that are found in many places in distributed systems. For example, a directory in a conventional filesystem is a group of filesystem objects which may, in turn, contain other directories. In systems like AFS or Novell where security permissions on a directory affect all files in the directory, the grouping simplifies the administration of security by providing a single object that can be modified to change a group of objects. In any hierarchichal filesystem, the hierarchichal grouping in the directory tree provides a mechanism to move large groups of files around within the filesystem by moving directories. Another common example of groups is the user-group found in most filesystems. The user-group abstraction simplifies management by allowing a single access control operation to affect everyone in the user-group. Modifying a single group data structure is both more efficient and less error prone than individually modifying a large number of users.

Group capabilities complement basic capabilities rather than replace them because group capabilities are an inefficient mechanism for fine-grained control. Consider a filemanager enabling a write request in AFS: a filemanager needs to grant access to a *single* object for a short period of time while it prevents other clients from reading the object and monitors the object length to enforce its quota and cache consistency policies [Gibson97b]. In contrast, consider a filemanager enabling a read access: the filemanager could enable read access to *all* the objects within a directory using a single group capability. If both cases used the group capabilities, the filemanager would need to create an indirection object simply to enable access to a single object. A more efficient solution is to add a flag to capabilities that indicate whether the capability should be interpreted as a group capability rather than as a basic capability since their structures are otherwise identical.

When a drive receives a request, the drive must retrieve the indirect object and verify that the requested object is member of the group as shown in Figure 5-6. The drive retrieves the indirection object referred to in the group capability and verifies that the AV in the capability is current by comparing it with the current AV of the indirection object. If the AV is current, the drive checks that the object being requested is a member of the indirection object with binary search in *log(N)* time, as long as the group membership list is maintained in sorted order.

A filemanager can build indirect objects either statically or on the fly. The static approach is well suited for a filemanager with access control policies tightly bound to the underlying structure of the filesystem. For example, in both AFS and AppleShare, access control is administered on a per-directory basis so it is natural for a filemanager to construct indirect objects that mirror this structure. If these indirect objects exist on the drive when a client requests an access credential, then the filemanager does not need to send the drive a sequence of requests to build the indirect object. However, when a client builds the indirect object in direct response to a client operation, the filemanager can exercise a finer degree of control over what operations it allows. The filemanager can maximize the rights granted to a client by building a large indirect group or enable access



Group Capability: Drive Identifier, Partition Identifier, rights, *AV*, *GroupA*
Request: (*Object1*, Drive Identifier, Partition Identifier, Offset, Read)
GroupA,
AV
Object1,
Object7,
..,
Object3212
Are the AVs equal?
Is *Object1* listed?
Load *GroupA*

Figure 5-6 Evaluation of a Group Capability

This graphs shows the basic tests and flow necessary for an indirect group capability system. A request specifies the object, Object1 in this example, that the client is attempting to access while the group capability contains a description of the bearer's access rights as well as the name of an indirection object, GroupA. The drive retrieves the GroupA, potentially accessing the media, and verifies that its AV is the same as the one specified in the capability while also checking that Object1 is listed in GroupA. If all checks are successful, the client is granted all the privileges described by the capability with respect to the GroupA.

to the minimal set of objects required for the task at hand rather than granting the client unnecessary privileges [DoD85].

With static groups following the directory structure, group capabilities can reduce the number of times a client must obtain a capability to the number of directories that the client touches. The drawback of the static approach is its reliance on an underlying structure in the application's layout of objects to essentially hang the groups off and the fact that static groups consume space on the drive to mirror these structures. Neither of these drawbacks are show-stoppers because most applications have some underlying structure to efficiently manage both the data and its security policy. Additionally, I expect the space consumed to represent these structures, which mirror indirect objects, will at most double, and relatively small compared to the amount of data stored. Otherwise the system is already making poor use of its resources.

Group capabilities are unable to capture dynamic relationships among objects. Dynamic inheritance can be implemented more efficiently with group capabilities because revocation involves revoking access through all the indirect objects rather than through all the objects themselves, which I expect to be significantly larger if the groups follow an underlying structure of the filesystem. However, this still requires touching all of the groups, rather than objects in the normal capability case, that are affected by an access control change. Of course, an application could be pathologically bad and use indirect objects to capture all sets of objects with size *k* which would result in more indirect objects than actual data objects, although they would all be small.

To provide the group abstraction to the filemanagers and clients, the drive must introduce a layer of indirection through another object which may require an additional I/O operation per request. Of course, this I/O operation may frequently hit in the drive's data cache after an initial access, but there will always be cases where the workload has flushed an indirect object from the data cache. As I explained in Section 5.1, an additional I/O can have a significant performance penalty. In order to get the dynamic dependency behavior, the additional I/O is necessary penalty to evaluate the dependency. However, a better solution would be a mechanism where indirection through another object was an explicitly invoked option rather than a requirement of the group abstraction. This would avoid the potential problem of additional I/O operations unless the filemanager explicitly uses the dependency feature.

One advantage of the indirection layer in groups is efficient revocation of rights to the entire group, although this advantage would seldom be used. By changing the AV on the group, the filemanager can revoke all rights granted through the group which could affect a large number of clients and NASD objects. However, the PDL AFS '99 workloads show that **StoreACL** operations, which may also be granting access rather than revoking it, make up only 0.0034% of the requests so access revocation on a directory wide basis is rare. The **StoreAttributes** RPC potentially changes the mode bits or owner information on a single object so it may also revoke client access rights to a single object. **StoreAttributes** make up 2.2% of the requests. Using the worst case assumption that both **StoreAttributes** and **StoreACLs** are revoking access permissions, revocation is still very rare and usually only affects a single file (most of the revocations would be from **StoreAttributes**). In NFS, only 0.3% of the operations are **AttributeWrites** which are potential mode changes to a file. There, in all the workloads studied, revocation is a rare operation and should not be used to motivate optimizations that may penalize more common operations.

5.4 Metadata Filter Credentials

In this section, I describe an approach which exploits and encourages an application's ability to embed application-specific security information within a NASD object's metadata. The NASD interface includes an un-typed array of bytes, 256 bytes in the prototype, called *fs-specific* in which application writers are able to store arbitrary object metadata. For example, the NFS on NASD prototype uses the fs-specific field to store a file's owner's UID and group ID as well as the Unix mode bits. An AFS implementation could store a unique identifier for a file's directory or an ACL in the fs-specific field.

At a high level, the filemanager provides clients with access credentials that test a requested object's metadata for the presence or absence of some feature which determines whether the access credentials are appropriate for a requested object. What is an appropriate mechanism to describe the required features of an object's metadata? The two strongest options I found were regular expressions or executable filters. An executable approach is more appropriate for following reasons:

- Regular expressions are not space-efficient at capturing some if-then-else relationships. For example, with a regular expression, it is awkward to express the normal Unix behavior of disallowing access to the owner based on the user mode bit even if all other users are allowed to read the object.
- With sufficient space, a regular expression could capture any property of the object metadata but an executable model is frequently more intuitive.

I have implemented a derivative of the Berkeley Packet Filter (BPF) [McCanne93] — which I call metadata filters — which allows filemanagers to define tests on object metadata. I used the BPF as a starting point because the tasks of a packet filter and the metadata filter are in some respects very similar. A packet filter is responsible for differentiating between a packet that meets some criterion and should be forwarded to an end-point and a packet that should not be forwarded. Analogously, the filter in a metadata filter access credential must distinguish between objects that the access credentials is applicable to and those that it is not.

I replace the object ID and AV in a capability, which explicitly name an object, with a metadata filter, which implicitly defines a set of objects. These improved access credentials are called *MF credentials*. The simple applicability test for capabilities, an equality check between the capabilities object ID and AV against the requested object's ID and AV, is generalized into the execution of the metadata filter against the requested object's metadata. Since the access credential mechanism no longer relies on a separate AV as part of the object metadata, the AV can be removed from the NASD interface. An application can readily implement its own AV-like behavior by storing an application level AV within its object's fs-specific field. However, MF credentials still include the partition ID in order to identify the proper key in the key hierarchy to use when generating the private credential.

Initially, as shown in Figure 5-7, when a drive receives a request from a client, the drive first performs some simple static checks on the filter to verify that it is safe to execute. Specifically, the drive verifies that no code or data references go outside of valid ranges for static references. Dynamically generated references through registers or memory must be checked at execution time. Potentially, the static checks could be avoided and done dynamically but a static analysis is more efficient because the filter may be reused if it is successfully cached and the static checks can be bypassed on cache hits. Similarly, statically checking a reference in a loop body will be less expensive than repeatedly checking it every iteration.

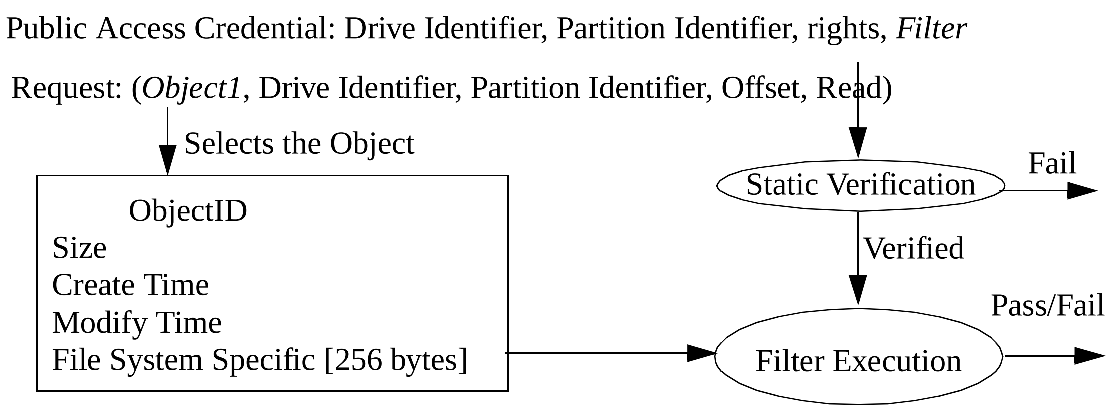

Public Access Credential: Drive Identifier, Partition Identifier, rights, *Filter*
Request: (*Object1*, Drive Identifier, Partition Identifier, Offset, Read)
Selects the Object
ObjectID
Size
Create Time
Modify Time
File System Specific [256 bytes]
Static Verification
Fail
Verified
Filter Execution
Pass/Fail

Figure 5-7 Evaluation of a Metadata Filter Credential

This figure shows the steps necessary to determine if a metadata filter credential is valid for a specific request. Each request includes a filter in the public access credential which the drive statically checks to verify that the filter can be safely executed. If the filter is safe to execute, the requested object's metadata is loaded and the filter is executed. When execution completes, the filter returns a pass or fail result that determines if the metadata filter credential is valid for the requested object. The processing time of the filter is limited by a time-out, set on a per partition basis, which prevents accidental infinite loop.

5.4.1 Size of Metadata Filters

A straightforward application of Berkeley Packet Filter technology to NASD results in filters that are only a few hundred bytes in size but waste space through unused instruction arguments. While exploring the size issue, I focused on three examples:

- Capability-like meta filters which check the object ID and AV. This filter was 80 bytes long.
- Filters that enable NFS clients to read any file with the appropriate mode bits for their user and group ID. This filter was 112 bytes long.
- Filters that enable AFS clients, who are members of four PTS groups, to read or retrieve the attributes of any file to which they are granted access, as long as no negative ACLs are listed. This filter was 248 bytes long.

The size of the filters is significant because every 64 bytes of access credential requires another iteration of the MAC function to generate the private credential, as well as the filter consuming space at both the client and the drive. If the filemanager or the drive is dealing with a burst of requests, the additional cost of generating the more complicated access credentials may be significant. None of these examples are extremely large, but I observed that the BPF instruction format, shown in Figure 5-8a, allocated half its space to a *k* operand that was frequently unused or used with the same value. This was an obvious opportunity to optimize away wasted space and reduce the overall filter size.

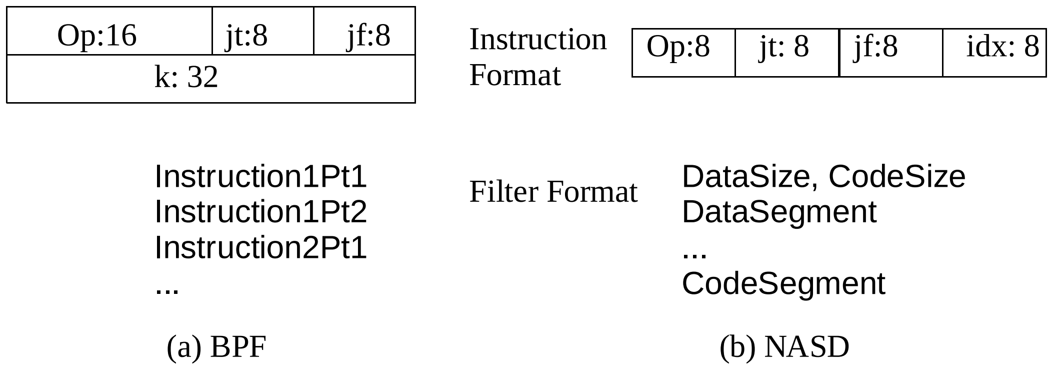


Figure 5-8 Comparison of BPF and NASD MF formats

In order to reduce the size of the metadata filters, I modified the Berkeley Packet Filter (a) to produce the NASD metadata filter (b). The BPF format is a sequence of fixed length instructions each containing a 16 bit op-code, two 8-bit offsets for conditional expressions, and a 32-bit multi-purpose field. A filter consists of a sequence of instructions. A NASD metadata filter replaces the multi-purpose field with an index into a separate data segment and compresses the op-code space to make individual instructions smaller. The abandonment of the larger multi-purpose field requires an additional data segment to be added to each filter to hold large constant values. The NASD metadata filter includes size information, a data segment, and a code segment which contains the shortened NASD format instructions.

By cutting the instruction size in half and separating the filter into a data and code segment, I was able to reduce the length between 25% (for the capability-like filters) and 38% (for the AFS read filters). As shown in Figure 5-8, an MF now contains a data segment that is used to initialize the filter's memory segment at execution time and holds constant data values. I replaced the 64-bit *k* operand in the BPF, which is frequently unused or partially used, with an 8-bit *idx* operand that references the data segment rather than including an immediate value. I was also able to cut the opcode size in half because it only used half of its allocated space, presumably because it was used to pad the overall instruction length to a 32-bit boundary. The instruction format changes save space on the control portions of the filter but not the parts that involve constants because they reference data in the data segment.

The smaller filters have the following size:

- Capability-like filters - 60 bytes
- An NFS filter that allows a user access to any file she is permitted to read - 84 bytes
- An AFS filter that allows a user who is a member of four protection groups to read an object she is permitted to read as long as it has no negative ACL set - 156 bytes

These sizes show that filters can capture a large amount of the access control structure process in a very small amount of space.

5.4.2 Information Leakage

If the evaluation of the metadata filter takes place before, or simultaneously with, the verification of a request's MAC, as discussed in Section 4.4.4.1, a bad MAC error must take precedence over a filter execution that returns a failure result. If the result of executing the filter is revealed when the MAC is invalid, then the drive is releasing information that was not approved by the filemanager. An adversary could generate a filter that asks a question about an object's metadata such as "Is this a top-secret object?" which the drive would blindly answer before checking if the adversary was entitled to this information. If the adversary constructed filters to test for the presence of specific bytes in specific locations, an adversary could use this feature to read all of an object's metadata.

However, if a filemanager gives a client a metadata filter credential, and the drive verifies the filter before checking the MAC on the request, then the filemanager has given the client the right to have the filter evaluated over any object's metadata. If the filemanager is concerned with clients learning the contents of object's metadata, a filemanager should hand out only very limited access credentials that will not reveal secret information when a false result is returned.

5.4.3 Support for Dynamic Access Control Systems

By extending the BPF language with the **LOAD_OBJ_META** instruction that loads a specified object's metadata to the filter language, I have enabled filemanagers to implement dynamic dependencies between different NASD objects for access control decisions. The default behavior, when the **LOAD_OBJ_META** instruction is not used, is for the filter to access only the requested object's metadata to make access control decisions, which do not require any additional I/O operations beyond those required when no security is used. By adding the **LOAD_OBJ_META**, the drive may require additional I/Os to service a request but only when the instruction is used rather than on every request as required when using indirect groups.

This feature allows applications to implement an access control scheme that logically walks up a directory hierarchy and evaluates access control at each step along the way such as in AppleShare, Novell, or Windows NT (with bypass traversal checking disabled). When a directory's permissions change, the filemanager needs to modify only the actual directory rather than all the effected filesystem objects, which is necessary in approaches without support for dynamic relationships.

Introducing dependencies on other objects complicates the caching of access credential. In all the other types of access credentials I have presented, the current validity of an access credential depends only on a single object. It is simple to invalidate cached access credentials when that object is modified. With metadata filter credentials, the validity of a key for a specific object depends on a set of different objects. These dependencies need to be tracked, probably by a hash table, so that out-of-date decisions will not be reused.

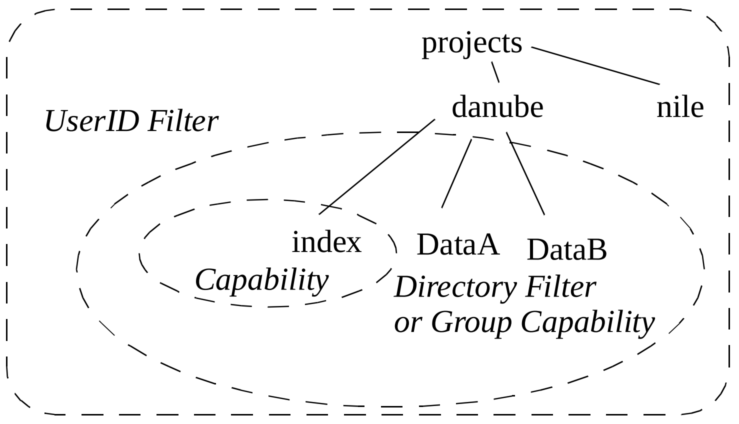


Figure 5-9 Example of Scope of Different Forms of Access Credentials

A capability enables access to exactly one object. A directory filter or group capabilities based on directories can enable access to all objects within a directory. UserID filters parse some of the ACL and/or mode bits and allow a user access to all the objects to which she is entitled.

5.4.4 Reduced Load on Filemanager

Metadata filter access credentials can capture a bigger chunk of the application access control semantics within the access credentials, rather then leaving all of the decision processing to the filemanager. For example, in some filesystems, application designers can implement the access control system by labeling each object with its home directory and simply checking this label with the access credential. Alternately, metadata filters can perform a portion of the access control check at the drive and a filemanager can issue a smaller set of metadata filter credentials to clients allowing clients to access all or some of the objects for which they have access permission. Both of these approaches capture larger chunks of a client's access space, as illustrated in Figure 5-9, and have a potential for greater reuse than capabilities.

Both AFS and NFS can be implemented using metadata filters that allow a user to access all the objects which she is permitted to read or getattributes. For NFS, the metadata filters implements the entire Unix access check for a fixed user ID and group ID. For AFS, a metadata filter for a read operation performs four checks:

1. Test for the presence of a negative ACL entry which will require the filemanager to become involved.
2. Decide if this is a directory or file object because different ACL entries are needed in each case.
3. Iterate through the ACL entries looking for one that grants permission to the clients PTS ID or one of the group PTS IDs listed in the access credential.

4. Check the mode bits on the object. As described in Section 3.5, this can be either the group or the owner bits depending on how the AFS server is configured.

Alternately, AFS on NASD could label each file object with an identifier indicating which directory it is stored in. Clients are given metadata filters that test for this label and only perform the mode bit check, the fourth step listed above, before granting a client access to the object. This makes sense for AFS because ACLs are shared across all objects in a directory. For NFS, access control information is on a per-file basis rather than shared across an entire directory.

I extended the simulations presented in Section 5.4.4 to study how metadata filter credentials can be used to reduce the offered load to the filemanager. For NFS, the filemanager issues a client two access credentials, one for reading/getattr and one for writing. For AFS, the filemanager issues a clients read/gettattr credential per directory, directory access credentials, or for all objects, user access credentials, as well as short-lived per-object write credentials in both cases.

For all the workloads examined, issuing access credentials based on directories or the user significantly reduced the offered load to the filemanager as shown in Table 5-1. For the AFS workloads, the impact was smaller because the AFS filemanager does more work than an NFS filemanager. The largest improvement comes from simply generating access credentials on a per-directory basis rather than per-object, although a small additional gain can be had by doing user-based access credentials. Initially, I expected the user based access credentials to get large gains in the AFS workloads just as Table 5-1 shows in the NFS workload. However, in all cases, the AFS filemanager is issuing short lived object write access credentials so that it can enforce its quota management and assist in cache consistency. This accounts for most of the access credential request traffic when user credentials are employed.

Table 5-1 Ratio of Load with Access Control Traffic at Filemanager Versus Without

For the three workloads, the table shows how close the different types of access credentials approach the load when the filemanager does not issue any access credentials. The metric used is the ratio of the offered load with the access credential requests being serviced by the filemanager to the load without the access control requests (i.e. the clients generate their own as described in Section 5.1.3). A ratio of 1.0 would indicate that the filemanager sees no additional requests to issue access credentials.

Type of Access Credential	NFS	AFS 1	AFS 2
Capabilities	2.75	1.79	1.35
Directory Access Credentials	N/A	1.19	1.09
User Access Credentials	1.01	1.13	1.07

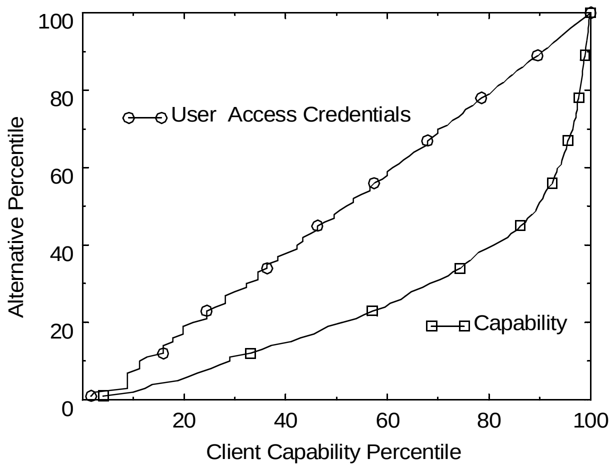


Figure 5-10 NFS on NASD Load Percentiles

User identity based access credentials eliminate most of the bowing of the curve, bringing the curve closer to linear which indicates that the offered load to the filemanager is similar to the load offered when clients generate capabilities. The y-values are the percentage of active client minutes with filemanager generated capabilities that are less than or equal to the maximum of the x-value's percentage most idle of the client-generated capability minutes.

Figure 5-10 and Figure 5-11 show the percentiles of the load on minute samples for client-generated capabilities versus the filemanager-generated capabilities, directory access credentials, and user ID access credentials. The graphs show that for both the AFS and NFS workloads, moving to an access control mechanism with greater expressiveness reduces the load on the filemanager so that the load with access control approximates the load with client-generated capabilities. This allows a comparable filemanager machine to be used regardless of whether the filemanager is administering the security policy of the system or trusting all clients.

Figure 5-12 shows the same 120 minute samples as Figure 5-3 and illustrates how more expressive access credentials can minimize the degree to which access control requests amplifies the burstiness of the workload. For both workloads, user ID based access credentials reduce the access control traffic to an extremely small delta over the best case of client generated capabilities.

5.4.5 Improved Drive Key Cache Hit Rate

Using metadata filters credentials, the access credentials capture more of the locality inherent in a workload which is evident in a higher hit rate if the access credential to key mapping is cached at the drive. In Section 5.1, I explained how computing the cryptographic keys for an access credential can account for a large portion of the cycles on a drive without hardware support. Based on the traces, I simulated the on-disk caches of

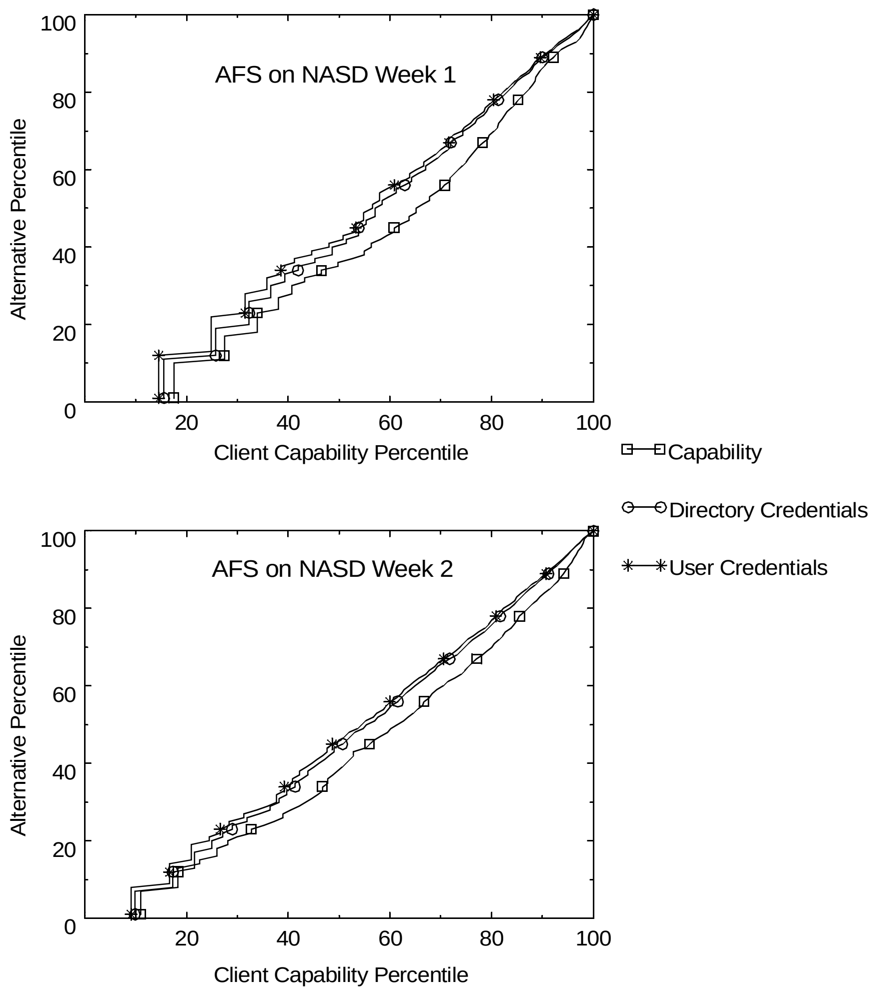
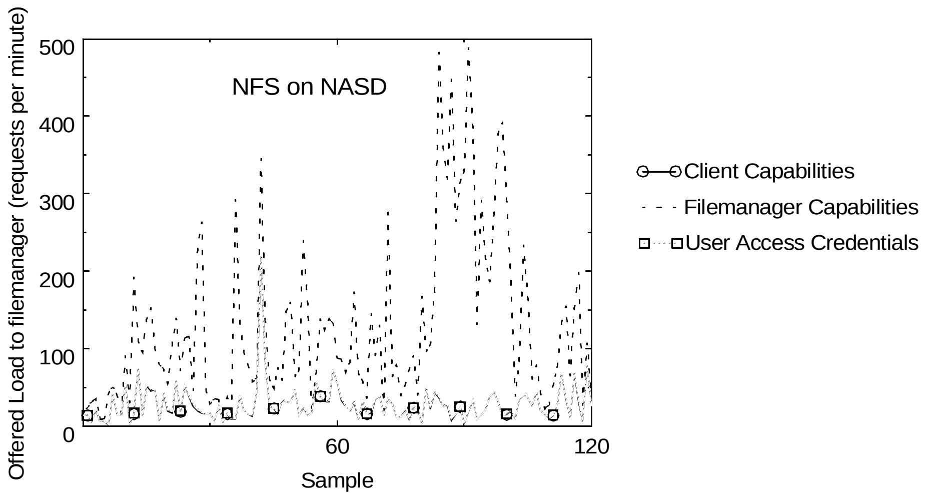


Figure 5-11 AFS on NASD Load Percentiles

Directory access credentials and user identity based access credentials provide a noticeable improvement over capabilities, although capabilities were already quite good for AFS since write capabilities and synchronization operations limited the impact of requests for read capabilities. The y-values are the percentage of active client minutes with filemanager generated capabilities that less than or equal to the maximum of the x-value's percentage most idle of the client generated capability minutes. There is a small but noticeable difference between the percentiles for client generated versus filemanager generated capabilities as shown by the distance of the curve from linear. With directory access credentials or client identity access credentials, the difference is much smaller.



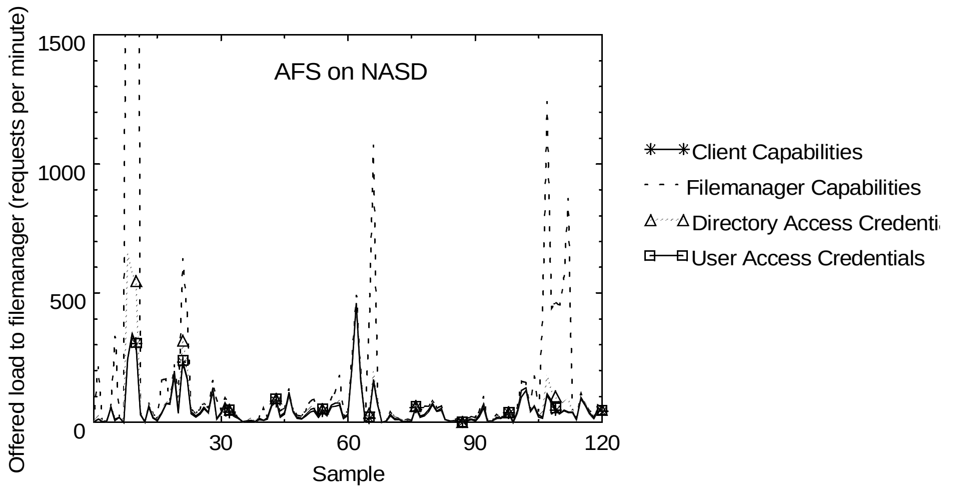


Figure 5-12 Filemanager Load Percentiles: Alternative Access Credentials

Both identity- and directory-based access credentials reduce the burstiness of the workloads and bring the load closer to the best possible case of clients generating capabilities. These graphs show the same 120 minute segments shown in Figure 5-3 with all the options for access credentials that were studied in Section 5.4.4.

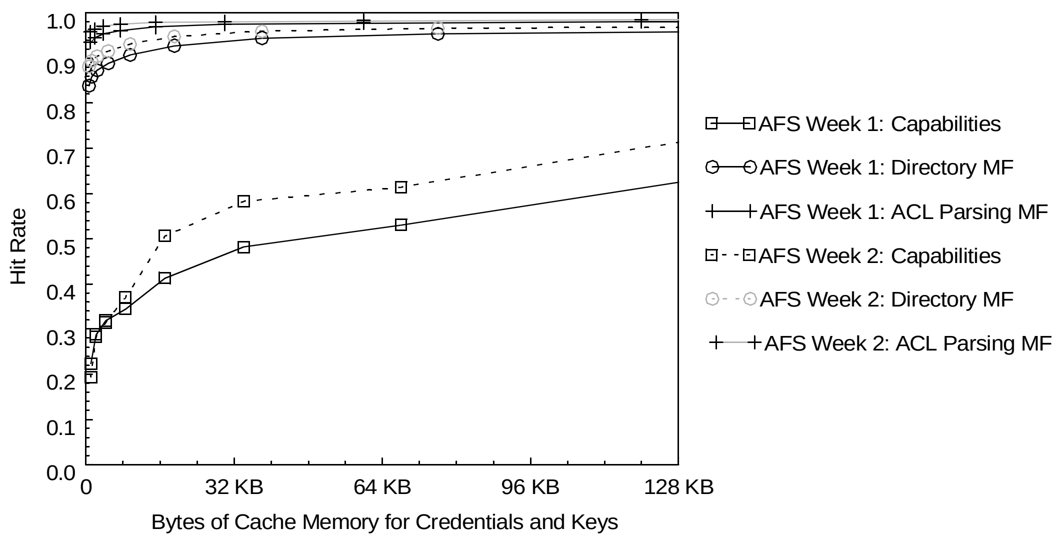


Figure 5-13 Hit Rate of on Disk Access Credential to Cryptographic Key Cache

Metadata filter access credentials can be cached more efficiently than capabilities. This figure shows the hit rate using two AFS workloads for a cache on a NASD drive that maps the public portion of an access credential to the MAC and encryption keys. All three approaches, capabilities, directory MF, and ACL parsing MF, were simulated with an LRU cache. In all cases, clients are given unique short-lived capabilities for write operations in addition to their read access credentials. In the capability case, clients are given capabilities that enable read and getattr access to a single object. In the directory MF case, clients are given MF access credentials that enable read and getattr access to all objects within the directory. In the ACL parsing approach, users are given access credentials that parse the ACL of the object and allow them to read and getattr all files or directories which they are allowed to access. All three approaches store 40 bytes of key per cache entry and the capability, directory based MF, and ACL parsing MF use 48, 112, and 200 bytes respectively for their public credentials. The MF approaches replace the AV and the object ID in the capability with an appropriate metadata filter. The size reported on the graph ignores overhead of maintaining the various data structures necessary for quick lookup and revocation.

the cryptographic keys to understand how the metadata filter credentials, which are larger than capabilities, impact the amount of memory necessary for the key cache.

Figure 5-13 shows that, as most computer scientists would expect, directories capture locality more efficiently than object identifiers, and user IDs are more effective than directories. This shows that the expressiveness of metadata filters enables a filesystem to implement its access control system in such a manner that it increases the hit rate in the on-drive key cache which reduces client latency. If the cache is more efficient, less valuable on-chip SRAM needs to be dedicated to caching keys, which can either reduce the drive’s cost or make the resources available for other functions. However, if a drive has an abundance of MAC bandwidth, then the savings of caching keys over recomputing keys will have a minimal impact on the performance seen by the client.

5.4.6 Costs

When using metadata filters, there are some additional costs that must be considered. Access credentials become larger, as discussed in Section 5.4.1, although they were only a couple of hundred bytes in the examples explored. Executing a metadata filter is more complicated than checking a few arguments in a capability so there is a higher computational cost to determine whether an access credential is valid for a particular object. Metadata filters also require space in an object's metadata to store information that the filter will check, so there is additionally a minimal metadata space requirement.

An executable approach such as a metadata filters requires more work by the drive to determine whether an access credential is valid for a specific object. With a capability, the check was simply a few instructions to check several fields in the requested object's metadata. With metadata filters, the drive performs a static verification of the filter to ensure that it is safe to execute, in addition to executing the filter. In the three examples I studied on a DEC Alpha 21064-based prototype, the static verification required between 310 instructions (capability-like filters) and 578 instructions (AFS identity based filters). Executing the filter required between 312 instructions (capability-like filters) and 760 instructions (AFS identity based filters). The verification and execution costs are significantly higher than the few comparisons necessary to check a true capability but are small relative to the 6000 cycles necessary to generate the MAC key for a request. This implies that the caching of metadata filters is an advantage when MAC computation is expensive. Additionally, the verification and execution costs of metadata filters would increase the total instructions executed for a request by 4% for a 1 B warm cache read, the data moving operation with the fewest executed instructions, and less than 1% for a 64 KB read or write [Gibson98].

Other techniques such as Software Fault Isolation [Wahbe93] and PCC [Necula96] may provide even better performance. SFI can improve execution time of a packet filter by a factor of four and PCC can improve execution by a factor of 10 [Necula96]. Both of these approaches incur a much larger (by a factor of 100 or more) fixed overhead to verify that a filter is safe to execute, but have a much lower per-instruction runtime cost. These higher verification costs can be amortized across multiple requests if the access credentials are successfully cached and reused. However, the high cost of verification will be on the critical path of the initial request using the access credential.

All three metadata filter examples require space in the metadata of the objects to store information that the filters examine but none require more than 180 bytes. The metadata filter works most efficiently operating on 4-byte values so every distinct value used in the test filters consumed 4 bytes of metadata. The capabilities used only the AV consuming 4 bytes of metadata. The NFS identity-based metadata filters need the owner's userid, owner's groupid, and file's mode bits to be stored in each object, which consumes 12 bytes of metadata. The AFS identity-based metadata filters stored an entire ACL in the metadata as well as the normal Unix security information that was used in NFS. An ACL was represented as the number of positive entries, the number of negative entries, and up to twenty ID-rights pairs. In total, AFS required 180 bytes in the metadata to represent all the necessary security information.

5.5 Policy objects

Experience with metadata filters shows that shipping functionality to the drive for security processes can provide performance wins as well as functional advantages. Observe that the policy in a given application seldom changes, although some parameters, such as a user’s identity, an ACL, a user’s group membership list, or an object’s metadata may change from request to request and from day to day. Instead of shipping the same metadata filter with slight variations to the drive with every request, I propose that a filemanager ship its policy to the drive in the form of an executable program and store this in an on-drive object, which I will refer to as a *policy object*. Essentially, the code segment of the metadata filter is being stored on the drive while the data segment is included in each client’s public credentials and sent with every request.

Policy objects have the advantage of not shipping data on each request that normally remains unchanged across requests from multiple clients. Only the data segment is included in the request which allows the public access credentials to be much smaller while preserving all the advantages of metadata filters. For capability-like functionality, the data segment includes an AV, partition, and object ID which total 16 bytes. For NFS identity based credentials, the data segment only requires 8 bytes to represent the user’s ID and group ID. For AFS, the data segment only requires 4 bytes to represent the user’s ID plus 4 bytes per protection group the user belongs to. This is a factor of 10-20 less space than required by the metadata filters reducing the communication overhead of shipping access credentials and decreasing the memory requirements for access credential caches. While the examples I gave for metadata filters are all relatively small, placing more complex behavior in the filters such as a full implementation of Files-11 semantics would require larger metadata filters and benefit more from policy objects.

The critical new concern created by policy objects is preventing malicious clients from convincing the drive to execute unauthorized code as a policy object. The drive must verify that the MAC on a request is valid for the given request and access credential, as well as making sure the access credential grants the rights necessary for the operation. When the drive verifies the MAC on a request is valid, the drive can conclude that the filemanager gave the client the access credential. Therefore, the drive can also conclude that the filemanager authorizes the drive to use the object referenced in the access credential as a policy object, i.e. the object is “safe”, as far as being in accordance with filemanager policy, to execute as a policy object. If execution preceded the MAC verification, then a malicious client could create a public credential referring to an arbitrary object and the drive would execute the object’s contents as a policy object, which could violate an application’s security policies. If the execution environment allows only read access, which is true in the filter model I presented earlier, then MAC errors must take precedence over filter execution errors to prevent information from being leaked by a client executing an arbitrary object. If the execution environment allows policy objects to have write access, the MAC verification must occur before the policy object is executed to prevent malicious modification of other objects. Alternately, NASD could prevent a client from executing an arbitrary object by including a **PolicyObject** flag in the NASD object metadata structure and only allow the filemanager to update this flag. This would allow the drive to verify whether the object reference in the public credential was intended by the filemanager as a policy object. However, this solution would also allow malicious clients to access different policy objects from the ones a filemanager authorizes, effectively making all policy objects into public oracles that clients can apply to arbitrary objects. Therefore, the precedence of the error codes is the proper solution to preventing malicious clients from convincing the drive to execute unauthorized code as a policy object.

With policy objects, the high cost of verifying the safety of an SFI or PCC filter is less of a concern. If the NASD interface includes the **PolicyObject** flag, the drive can perform the expensive static checks of an SFI or PCC filter when the filter is written rather than at execution time. This allows the drive to take advantage of the lower per-instruction cost of SFI and PCC relative to Berkeley Packet Filter derived approaches. Using the **PolicyObject** flag complements, rather than replaces, the precedence of MAC computation over filter evaluation.

5.6 Related Work

Many filesystems provide the kinds of dynamic behavior that many of my alternative solutions attempt to capture. For example, Novell [Sheldon96], AppleShare [Apple98], and Windows NT [Frisch98] all have mechanisms to check permissions involving an object and its ancestors rather than simply checking at the node being accessed. However, all these are filesystem specific ideas with different interfaces while NASD needs to have a more general mechanism that can emulate the mechanisms in these types of filesystems.

Some existing filesystems support some mechanism for aggregation which are similar to the grouping mechanisms discussed in this chapter. The explicit naming of NFSv3’s ReadDirPlus [Callaghan95] or AFS’s BulkStatus [Transarc91] operation are examples of this behavior. In this chapter, I explored some similar mechanism based on explicit naming but they fail to capture the dynamic behavior that would also be beneficial for NASD because they are explicitly naming objects rather than implicitly describing a group of objects. This is one of the reasons why the flexibility of the programmable solutions is appealing for NASD.

The idea of policy objects builds on a long history of similar ideas that allow an application to define functions that are evaluated on every request to enforce security policies. The earliest example of this type of function was the Multics’ TRAP function [Daley65] which associated a function with an ACL entry. More modern systems, such as Bershad’s Watchdogs [Bershad88] and Rabin and Tygar’s ITOSS system [Rabin89], allow functions to be inserted on the I/O path for a filesystem object, which are invoked on every operation to the filesystem object. In Chapter 3, I discussed how this type of function and the fine grained security used in DBMS systems posed a challenge to NASD. When an application designer can install an arbitrary function on the data path, the application designer can provide the behavior of the TRAP-like systems as well as the fine grained access control that DBMS systems normally provide.

5.7 Conclusion

Capabilities are a simple security mechanism that is efficient at capturing simple static access control decisions. Its simplicity makes it easy to implement with no extra I/O operations. However, capabilities are less than ideal. Limited by their simple explicit naming of objects, a capability cannot capture any dynamic relationships between storage objects. Requests for capabilities may substantially increase latency seen by clients. Additionally, a significant portion of the filemanager's load and the burstiness of the load may be directly attributable to clients requesting capabilities. Finally, the explicit naming of objects in a capability makes it impossible to use a single capability for an operation involving multiple objects and limits the efficiency of a cache of the private portion of the capability.

Moving from a simple capability model to a richer metadata filter model enables multi-object operations, better cachability, reduced client latency, and reduced offered load to the filemanager. All of these benefits arise from metadata filter's ability to ship portions of the applications access control policy to the drive. When the scope of individual access credentials is increased, the clients need to obtain them less frequently, which avoids latency penalties and load on the filemanager. Since the increase in size of the access credentials is significantly smaller than the increase in their scope, they become much more cachable which is beneficial in systems where the cost of generating the cryptographic keys is high.

Going further, policy objects allow a filemanager to install portions of its access control policy directly into the drive. This delivers all the advantages of metadata filter credentials as well as enables a finer degree of control by the filemanager. When security policies are installed at the drive rather than shipped with each request, the drive can use more aggressive technologies such as PCC or SFI to provide faster execution environments without suffering from the higher costs of these technologies' initial static verification.

Chapter 6: Efficient Drive Protection of Communication Integrity

Protecting the integrity of data transferred between clients and drives is one of the primary security concerns for network attached storage, as discussed in Chapter 2, and it is also computationally expensive. Although the network is untrusted, clients and storage devices must communicate safely without a malicious adversary being able to tamper with the messages. As I will show, protecting against these concerns normally requires more computational capacity than I expect in early NASD drives. This chapter explores alternative secure methods for protecting the communication integrity of read and write operations — the operations that move bulk data — to a storage device which can be implemented in the available resources of a NASD.

First, I describe the limitations of a conventional software solution implemented in a prototype drive [Gibson98] to motivate alternative solutions and establish baseline performance. Next, I describe an alternative approach, called “Hash and MAC”, that significantly reduces the cost of protecting the integrity of read traffic in storage devices that are unable to generate a message authentication code at full data transfers rates. The key idea in “Hash and MAC” is to perform a little extra work on write operations to precompute security information. After presenting “Hash and MAC”, I discuss the security of this approach and evaluate its performance within our prototype drive and compare its performance to the baseline prototype, which implements the design presented in Chapter 4. Finally, I refine the “Hash and MAC” approach by using incremental hash functions which improve the performance of small read and write operations as well as non-block-aligned operations. The advantages of the incremental hash approach are evaluated through a Mathematica model because the networking performance in the prototype would hide the gains of an incremental hash in the prototype.

6.1 Limitation of Software Cryptography

Software cryptography can limit the performance of a network attached storage device if insufficient CPU cycles are available. In order to explore the performance implications of security on the network attached storage architecture, I have implemented the basic NASD security system described Chapter 4 within the CMU NASD prototype. I approximate the expected computational capacity of early network attached storage devices by running the prototype on DEC Alpha workstations (using 133 MHz 21064

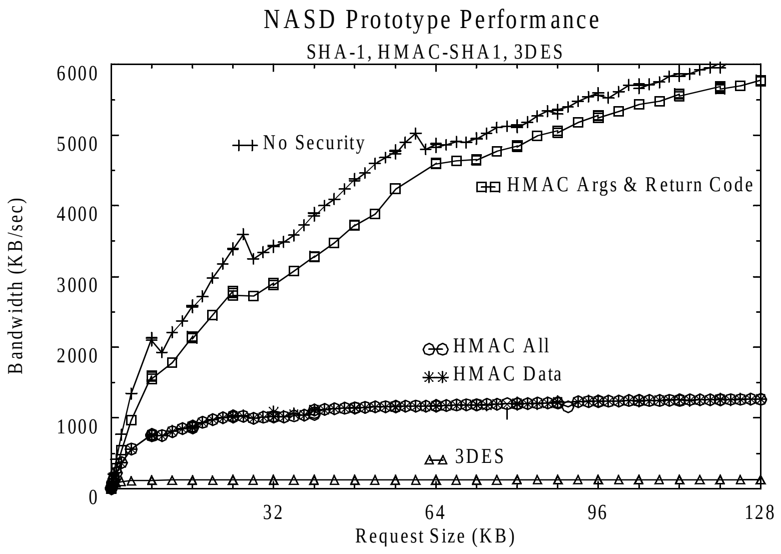


Figure 6-1 Prototype Read Bandwidth

Protecting the integrity of the arguments and return code imposes a small fixed performance penalty, while protecting the data exacts a much higher cost and saturates the drive CPU. This graph shows the results of a read microbenchmark of a prototype drive. The x-axis shows the size of requests made by the client and the y-axis marks the average read bandwidth. Each point represents read throughput for a minimum of 3 seconds of continuous requests and a minimum of 100 requests by a single client reading data from an in-memory object at the drive.The large irregularities in the shown on the graph are an artifact of using DCE for the RPC layer.

processors that were introduced in 1992) which are 3 generations behind current systems [Gibson98]. Drives communicate over an OC-3 ATM DEC Gigaswitch to clients, which are 233 MHz Alpha workstations. As described in Section 4.4.4, I use HMAC-SHA1 for integrity, and 3DES for privacy, as the low-level cryptographic primitives to implement security. Using this prototype of future network attached storage systems, I studied the impact of software cryptography.

Figure 6-1 shows the NASD prototype's performance across a range of software-implemented security options. For most security options, performance increases as we increase transfer sizes. With no security, which is an indicator of the raw performance available from the prototype, the performance curve flattens at around 6 MB/second as the drive CPU saturates and becomes the bottleneck. Most of the drive's cycles are not spent in the high-level NASD functionality but rather are spent in the RPC layer of the prototype which illustrates the importance of light-weight RPC implementations for network attached storage [Gibson98].

The highest-performance security option is **IntegrityArgs**, which protects the integrity of the nonces, request arguments, such as the object identifier, byte-range, and other operation-specific fields, and return codes, but does not protect data. This level of security permits control over what operations are performed but does not prevent an adversary from tampering with the data payloads. With this level of protection, the amount of cryptographic work is a function of the request arguments and return code size and is independent of the size of data transferred. Thus we have a fixed cryptographic overhead that is amortized over the size of the data. As shown in Figure 6-1, for 1 KB reads, the fixed penalty reduces performance by 30% while for 128 KB reads performance is reduced by only 7%. However, **IntegrityArgs** provides a weak notion of security because data is not protected from an adversary.

More protection can be achieved by using both **IntegrityArgs** and **IntegrityData** which together protects the data, return the integrity of data, arguments, and return codes at the price of a fixed cost per-byte. For non-trivial request sizes, the per byte cost of cryptography dominates the fixed cost of protecting the arguments and return codes. Figure 6-1 shows that the prototype maximum throughput is reduced by 46% for 1 KB reads and over 65% for 8 KB reads. At an 8 KB request size, the CPU saturates due to cryptography; larger transfer sizes do not provide any performance improvements.

Another important security option, **PrivacyData,** reduces performance by an even larger margin. Privacy of all the data, which has a high fixed per-byte cost, reduces performance to 126 KB/sec regardless of whether or not any integrity protection is used. Other algorithms may be faster in software but not by the factor of 240 necessary to meet modern 30 MB/sec disk drive media rates, so the problem can be expected to remain. Current efforts to define a new encryption standard, the Advanced Encryption Standard [NIST98], are focusing both on security and performance criteria to define a new encryption solution. The selection process may produce a freely-available and well-studied encryption algorithm that is designed for both high-performance hardware and software implementations. AES is unlikely, however, to provide the factor of 240 performance improvement essential to high-throughput encrypted communication on low-cost devices.

The cost of cryptography in software limits NASD performance except when using the weakest of security protections. For non-trivial data transfers, any security protection of data bytes reduces available bandwidth by 65% or more. Therefore, other approaches are necessary to provide high-performance and security from a low-cost storage device or the storage device will require significantly more computation than a 133 MHz Alpha 21064.

6.2 More Reads than Writes: An Opportunity

In many networked storage systems, applications read more data than they write [Baker91]. Many data sets change infrequently but will be read repeatedly, possibly by multiple users, before the data is changed. Good examples of this behavior are executable files, data mining databases, mail files, directories, and news files. The Berkeley NFS and PDL AFS traces studied in Chapter 5 both support this observation.

The Berkeley NFS trace shows a read-to-write request ratio of 4.8 to 1. Aggressive client caching employed by filesystems like AFS reduces the read traffic. The caching can also absorb write traffic to short-lived files because the files are deleted before they ever leave the client's cache. In the PDL 1999 AFS workload, I saw a read-to-write request ratio of 1.7 to 1. This creates the opportunity to reuse work across the multiple read operations.

When data in a file changes infrequently, then so does the value of functions of the data such as checksums. One optimization that exploits this unevenness between read and write frequency is to *precompute* network checksums across a set of data blocks and store each block's checksum information with the original data. When a read command is processed, the data and checksums are read from disk (or cache RAM), thus avoiding the cost of on-the-fly computation. Previous webserver research has shown that OS support for specialization of the web server, including avoiding checksum calculations through precomputation, can improve web server throughput by a factor of 2.3 over the Harvest httpd accelerator acting as front end to an NCSA server [Kaashoek96]. This suggests that a similar optimization may be useful for storage systems. However, there is a problem applying this optimization to a storage system. In web applications, documents are requested in their entirety while storage requests are frequently for portions of a file. It is impractical to store checksums for all possible client requests.

Additionally, a checksum does not provide any security guarantees. A checksum is designed to detect, and sometimes correct, a random error. An adversary can easily change a message without changing the value of the checksum. However, a checksum is very similar to two security tools that are used to provide integrity: message digests and message authentications codes, which I will discuss in the next section.

6.3 Communication Integrity

Communications integrity is the ability of the receiver to know that the data received was not modified by an adversary on the network during transmission. Two cryptographic mechanisms are generally used to provide communications integrity:

1. Message digests[1] with public key signature[2] (a.k.a. "Hash and Sign"), and
2. Message authentication codes (or MACs which were used in the baseline prototype described in Chapter 4).

1. A message digest (MD) is a strong checksum that processes a variable-length input string to produce a fixed-length output(e.g., MD5 [Rivest92], SHA-1 [FIPS180-1], and Tiger [Anderson96a]). While normal checksums are designed to detect random errors, a message digest detects deliberate malicious modifications by an adversary [Schneir96].

2. Public key cryptographic systems use one key for encryption and a different key for decryption. When the message is encrypted with the private key and decrypted with the public key, this usage is called a *signature*. Anyone who has the appropriate public key will be able to verify the signature of a message [Schneir96].

Message digests, also called cryptographic hash functions, alone are insufficient to provide communications integrity. A message digest is essentially a cryptographic fingerprint of a message. If an adversary changes a message, the modified message will generate a different message digest. However, since no secrets are used to generate the message digest, anyone can generate a valid message digest which permits an adversary to modify a message and replace its message digest with an updated version to avoid discovery. On their own, message digests are insufficient to provide communications integrity.

The “Hash and Sign” provides integrity by signing a message digest (i.e., encrypting) with a user’s private key. The process of signing, encrypting with the private key, binds the key to the signed digest so only the holder of the private key can generate the appropriate signed digest. However, anyone with access to the public key can decrypt and verify the message digest. Since the public key will only properly decrypt messages encrypted with the matching key, if the correct message digest is decrypted then the message must have originated from the holder of the private key. This approach also provides non-repudiation; the sender is the only one who has and has ever had the private key, unless the key is compromised, so nobody else could have sent the message. Unfortunately, public key encryption operations are generally 10x-100x more expensive than message digest operations, because of their reliance on number theoretic properties rather iterating a simple operation, making them ill-suited for performance-sensitive applications.

The second approach, using message authentication codes[1], is similar to a message digest except it uses a secret key in addition to the variable length input to produce a fixed length output. Using a secret key ensures that only holders of the secret key can generate or verify the MAC. In a hash-and-sign approach, the hashing phase does not involve a cryptographic key and the signing phase does. A MAC normally involves the secret key in the entire computation, so unlike the hash and sign approach, the computation cannot occur until the cryptographic key is known (thus we can not preform the precomputation optimization which I will discuss in the next section). These MAC approaches do not provide non-repudiation because both the sender and receiver share the same key. Thus an objective third party cannot differentiate between which of the parties sent the message. However, in many applications, such as NASD, we are already trusting the parties that distribute the keys so this is not a necessary feature.

A NASD drive will be a commodity device and sensitive to any increase in cost. Even a small incremental cost to a drive amounts to an impressive figure when you consider the number of drives shipped annually. For this reason, drive manufacturers are hesitant to add potentially costly security hardware to a drive. As a result, some drives will have a significant difference in their basic throughput, media rates and network interface rates, and cryptography throughput. In the remainder of this chapter, I explore some

1. Good examples of modern MAC algorithms are HMAC-SHA1 [Bellare96a], XOR-MAC [Bellare95], or MDx-MAC [Preneel95].

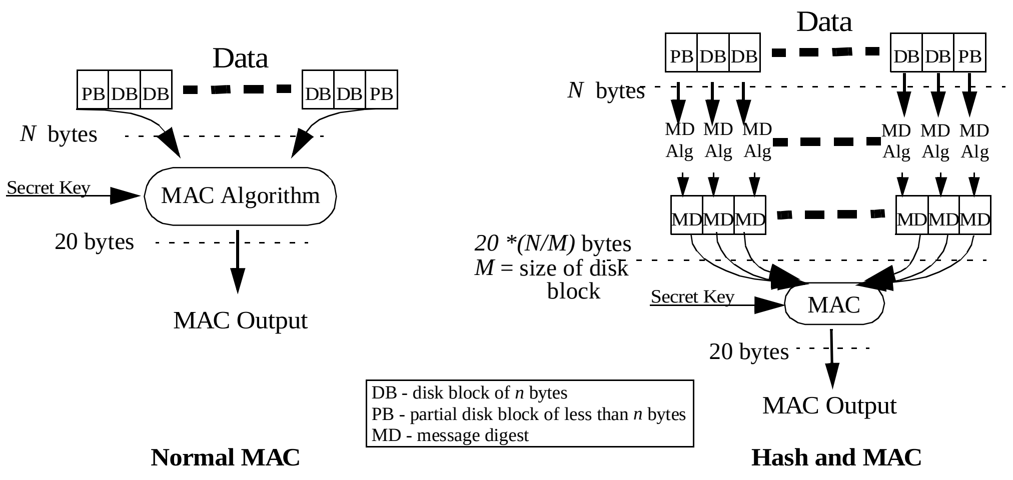


Figure 6-2 MAC Structures

The Hash and MAC approach reduces the amount of computation that involves the secret key. Each message consists of a sequence of full disk blocks which may be preceded and/or followed by a partial disk block. On the left, most MAC algorithms involve the key in the computation over all the bytes of data and process the data linearly. On the right, Hash and MAC does not involve the key until late in the computation. This enables parallelization and precomputation for increased performance. The labeled dotted lines indicate the amount of data that passes in and out of the message digest or MAC algorithms at the different stages of computation. In the Hash and MAC approach, a calculation over only 20 bytes per disk block involves the key while the rest of the computation can potentially be precomputed without knowledge of the key.

optimizations to provide high bandwidth integrity protection from a drive while using fewer resources than the software prototype performance in Section 6.1 would imply.

6.4 Hash and MAC

To provide strong integrity guarantees and exploit precomputed information as suggested by webserver research, I propose a different structure for protecting integrity of data that *explicitly* delays the binding of the key to the computation. Based on existing message authentication code and message digest algorithms, my approach, which I call "Hash and MAC", does the following:

- When a drive object is written for the first time, the drive precomputes a sequence of *unkeyed message digests* over each of the object's data blocks.
- For each read request to the drive, the drive generates a MAC of the concatenation of the unkeyed message digests corresponding to the requested data blocks.

Normal MAC algorithms (Figure 6-2a) involve the key throughout the entire computation of the message authentication code. In contrast, I remove the key from the per-byte calculation and use it only in the final step of the calculation (Figure 6-2b). Because the key is not involved in the per-byte calculations, the results of the per-byte calculation, a set of message digests, can be stored and used for multiple read requests to the same disk block from different clients. Additionally, since no key needs to be identified before a message digest can begin, message digest processing may be simpler for high-speed hardware than MAC processing, which must delay until the proper key is identified.

The “Hash and MAC” approach is very similar to encrypting or signing a message digest. However, it does not provide the non-repudiation that a public key system provides. In this sense, it is more like a normal MAC or like encrypting a message digest with a symmetric key system.

6.4.1 Security of Hash and MAC

How does the “Hash and MAC” approach effect the security of the system? MACing the concatenation of hash values is very similar to signing them with a public key except it is much faster and does not provide the non-repudiabilty property associated with public key signatures.

If we assume the basic MAC function is secure, is the MAC of hash values secure? When something is considered “secure”, it is normally secure for an arbitrary input. If there were a class of inputs for which it was insecure, then the MAC function as a whole would not be secure. An adversary breaks a MAC if she can recover the key or generate a MAC value for a message which she has not seen before. Concretely, if “Hash and MAC” is broken by attacking the MAC function, then a set of inputs, the concatenation of hash values, has been defined that can be used as an input to the MAC to break the original MAC. By our initial assumption, the MAC is secure so this can not be true.

An adversary could attack “Hash and MAC” through the message digest. “Hash and MAC” trades off some security in exchange for increased performance. An adversary can mount an off-line attack against the message digest function, essentially computing with no information about the message being attacked. With a normal MAC, an adversary could not start an attack until she was given a message to attack because the result of the key-dependent computation was essential to the attack. An adversary can apply arbitrary computational power to *precompute* two data blocks that generate the same digest (i.e., a collision). Alternately, an adversary who observes a series of requests and their associated message digests can attempt to find a second data block that generates the same digest as a given message block (i.e., a second pre-image). The difference is between the adversary being allowed to select both blocks in the collision as opposed to being given one of the blocks, which can be viewed as a challenge, and trying to find a second block which generates the same MAC. As long as NASD uses a strong message digest with a large output, such as SHA-1 or RIPEMD-160 which produce 160 bit outputs, the off-line attack is a small risk to NASD security. The best current attacks against these message digests requires a brute force search of the input space. In order to find a second pre-image of a given message digest, an adversary expects to compute digests of on average 2^{160} data blocks. A far simpler task, given large amounts of memory, is to find a pair of data blocks which generate the same hash by exploiting the birthday paradox, but this attack is still expected to require 2^{80} digest calculations.

Assuming an adversary is able to find a collision, she can exploit the collision if one of the colliding blocks is already within the storage system and the adversary can replace the in-system block with the out-of-system colliding block in a message exchange. An adversary can potentially tabulate a large number of digests and watch message traffic to the drive for an opportunity but the odds of such an opportunity presenting itself is:

$$2 \times \frac{NumOfCollisions \times NumOfBlocksSeen}{2^{160}}$$

An adversary will have an easier task if she can insert one half of a collision into the storage system and then replace it with the other half. In this case, she could have already written the second of the two blocks to the storage rather than swapping the blocks while they were being read. Thus, an adversary can primarily exploit a collision in a multi-tier system, such as a database system, where write operations are filtered through another host which decides if a write should be forwarded to the storage. If a collision is found, the adversary can swap a bad data block for the forwarded data block. Because the filtering host is making a decision based on the contents of the initial write request, it is implicitly enforcing some structure on the writes it forwards on to storage. Since one half of the collision must fit the required structure for the filtering host to forward it, this structure improves security by constraining the set of useful collisions an adversary can theoretically generate.

Because “Hash and MAC” generates multiple independent digests which are used to create the final MAC, an adversary can parallelize an attempt to find a second pre-image of the digests. If the request is divided into *r* different data blocks, an adversary can attack *r* different values when trying to find the second per-image of a digest. In contrast, a normal MAC algorithm has a single MAC value that can be attacked because all partially computed values are key dependent and hidden in the MAC algorithm’s internal state. Even for extremely large requests and heavily used storage devices, *r* will not be large enough to substantially reduce the 2^{160} computations required to find a second pre-image. For example, if a client transferred a terabyte of data and the digests were generated on 8K disk blocks, then an adversary could attack 2^{22} unique messages digests. This only reduces the work factor from 2^{160} down to 2^{138}. In order for parallelism to reduce the work of finding a second pre-image down to the work of finding a simple collision, the adversary would need to observe 2^{80} disk blocks and attack them in parallel.

To place these numbers in perspective, in 1999, a distributed computing effort over the internet was able to break a DES key which was expected to require 2^{56} calculations to search the keyspace in under 1 day [EFF99]. If computing power doubles every year, finding a collision (2^{80} calculations) will take 1 year in 2014 if enough capacity was available to store and search 2^{80} message-digest pairs. Under the same assumption, finding a second pre-image (2^{160} calculations) will take 1 year in 2094. An important caveat is that these numbers assume no technique better than brute force will be discovered to attack the hash function.

6.4.2 Performance of Hash and MAC

NASD's implementation of "Hash and MAC" uses SHA-1 to compute each disk block's message digest and HMAC-SHA1 for the overall message authentication code. I will refer to this specific instantiation of Hash and MAC as *HierMAC*. When data is read, the precomputed message digests are read from the drive and used as input to the HMAC-SHA1. If only a partial disk block is read, which can only occur in the first and last disk blocks of a request, a message digest of the partial disk block is computed on-the-fly.

With a normal MAC, the cryptographic costs were directly proportional to the number of bytes being transmitted. HierMAC reduces the cost to

$$RequestHdr + (PrefixBytes + SuffixBytes) + NumOfFullDiskBlocks \times DigestSize$$

where *(PrefixBytes + SuffixBytes)* are the number of bytes in the partial data blocks. In our implementation, a disk block is 8KBytes while a message digest is 20 bytes. Thus, HierMAC performs a MAC operations on 20 bytes per full disk block (8KBytes) transferred. This reduces (in the asymptotic case) the request time integrity processing to 0.2% of the number of bytes that a normal MAC would process. We are not changing the total number of bytes processed by the MAC algorithm. Instead, we are reordering the work in time and sharing work across multiple commands to reduce the on-the-fly cryptographic load.

Figure 6-3 shows that reducing the on-the-fly cryptography significantly increases read throughput of the NASD prototype but not as much as expected. For large requests in multiples of 8KBytes, protecting the integrity of all the data decreases performance by 45% percent of our maximal performance. In fact, with stored digests, the drive can transmit integrity-protected data faster than the client is able to verify the data's MAC, shifting the bottleneck from the NASD drive to the receiving workstation.

NASD drives must be inexpensive because they are high-volume products in a commodity market and there may be hundreds or thousands in an organization's infrastructure. In contrast, client machines are likely to have high-performance processors or dedicated hardware for special tasks, such as security, that are critical to their regular function.

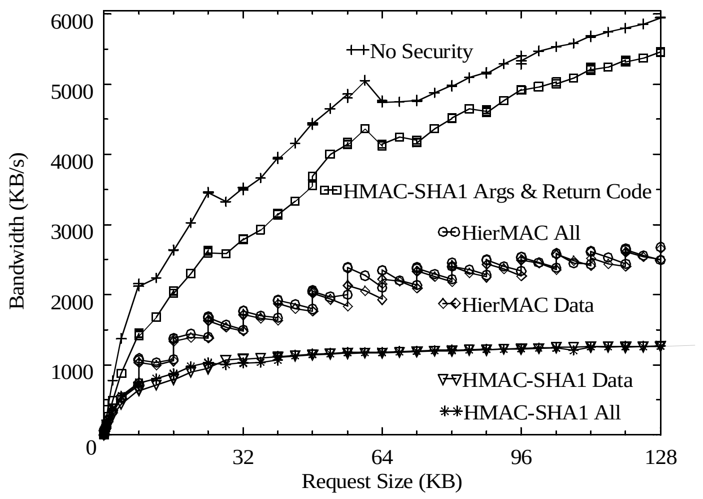


Figure 6-3 HierMAC Performance

HierMAC, based on SHA-1 and HMAC-SHA1, doubles performance when protecting the integrity of data for large requests when compared to HMAC-SHA1, also shown in Figure 6-1. Reuse of stored message digests has substantially improved the read bandwidth for large requests but fails to deliver the bandwidth we would expect given the reduction in cryptography at the drive because the client has now become the bottleneck. The x-axis shows the size of requests, which start at the beginning of an object, made by the client and the y-axis marks the average read bandwidth. Each point represents read throughput for a minimum of 3 seconds of continuous requests and a minimum of 100 requests by a single client reading data from an in-memory object at the drive. HierMAC stores an SHA-1 digest on each 8 KB disk block.

With a faster client, the drive will become the bottleneck. To understand the performance potential of HierMAC, I emulated a faster client by disabling the verification of the MAC at the client end. Figure 6-4 shows that HierMAC's low per-byte security overhead allows performance to closely follow the no-security performance curve. Total cryptographic costs are now small enough that cost of protecting the arguments plus the data causes a noticeable performance difference rather than being masked by the overlapping of computation at both ends of the communication.

Figure 6-4 shows that precomputing message digests on 8KByte blocks creates a pronounced saw-tooth behavior. Between 8Kbyte block boundaries, performance declines because the drive spends more time processing the prefix and suffix bytes from the partial disk blocks. On 8KByte boundaries, the drive uses only the stored digest (*prefix* + *suffix* length returns to zero) and the cost of protecting integrity is minimized. For a uniform distribution of starting and ending bytes within a file, the average number of *prefix* + *suffix* bytes will be the size of one data block. Thus, the performance at the lowest points of the saw-tooth, 1 byte before hitting a disk block boundary, will represent the expected average performance for a randomly selected read request. Many filesystems attempt to make requests that are aligned on disk block or VM page boundaries, which will result in significantly better performance.

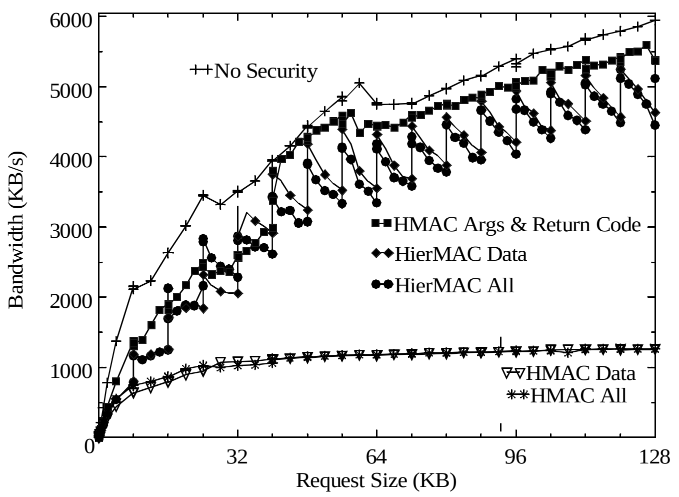


Figure 6-4 Fast Client with HierMAC

When the client is no longer the bottleneck, the drive delivers integrity-protected bandwidth close to the maximum bandwidth of the prototype. The saw-tooth behavior occurs as the drive generates an on-the-fly message digest of the partial final data block. The x-axis shows the size of requests, which start at the beginning of an object, made by the client. The y-axis marks the average read bandwidth. Each point represents read throughput for a minimum of 3 seconds of continuous requests and a minimum of 100 requests by a single client reading data from an in-memory object at the drive. HierMAC stores an SHA-1 digest on each 8 KB disk block.

Figure 6-4 shows that by closely integrating security with storage we have achieved a significant improvement in our read bandwidth. However, we gain no performance benefit on either writes or small read operations. For many workloads, the majority of bytes are moved in large requests, making HierMAC a powerful optimization.

6.4.3 Storing the Precomputed Message Digests in NASD

NASD uses an inode structure similar to the Berkeley FFS [McKusick84] which contains indirect and direct data block pointers as well as the stored digests. Figure 6-5 depicts how the stored digests are integrated with the direct pointers in the inodes. A significant advantage of storing the digests with the direct pointers is that whenever the data is accessed,[1] the digests will always be available without additional I/O operations. However, the addition of a stored digest to each direct pointer entry reduces the total number of pointers that fit in a single inode, thus reducing the addressable storage at any given level of indirection and the overall addressable storage of a single NASD object.

1. For drives, the most expensive operation is to physically move the arm to another location on the media. In a modern Seagate ST31903 drive, average seeks are 5 msecs [Seagate99a]. In order to avoid this penalty, we want to minimize the number of I/O operations necessary to service a client's request. If retrieving stored digests introduces additional I/O operations, we would lose some, if not all, of the performance advantage of stored message digests.

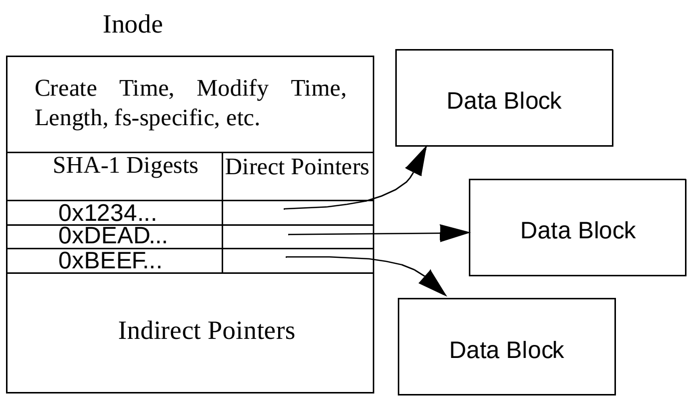


Figure 6-5 Storage of Digests in Prototype Drive

HierMAC's SHA-1 digests consume 20 bytes per director pointer in each inode. In order for the drive to service an operation on an object, the drive must read the object's metadata into memory in order to find the data blocks. I store the SHA-1 digests in the inode structure along with the direct pointers to the data so they are always available when we are accessing the data.

Within the prototype, the addition of the stored SHA-1 digests reduced the number of bytes addressable via direct pointers in an inode from approximately 4MB to 1.8 MB. Similarly, using our 8K inodes, the total addressable storage in a single NASD object is reduced from over 4.0 EB down to approximately 8 PB. Because the inode structure allows multiple levels of indirection, a different implementation may select a different balance of direct and indirect pointers to trade off direct addressability in an inode for the total addressable storage for an object.

Whenever a client writes new data, both the stored data and the stored digest must be updated. If they become mismatched, the drive will send clients erroneous MAC values and the clients will reject the data because it will believe that an adversary tampered with the data. This problem of keeping the stored digests consistent is very similar to the problem of avoiding errors with pointers to inodes and disk blocks in a filesystem, but there are some significant differences. Data block pointers are only modified when disk blocks are allocated (i.e., the file is extended) or deallocated (i.e., the file is shortened). In contrast, stored digests will be updated on every write operation, opening a bigger window for errors due to system crashes. Some filesystems also update a last access time on every request, but it is not critical to system integrity. Thus the filesystem does not apply aggressive techniques to keep it accurate when the system fails. Additionally, the last access time does not exist in indirect inode blocks.

A NASD drive can keep stored digests and disk blocks consistent in the presence of drive failures through one of the following mechanisms:

- A log of dirty inodes is stored in a small NVRAM. Before a disk block is written to the media, the appropriate inode is marked as dirty in NVRAM and the stored digest is updated. If the drive fails between receiving the write and having both the inode and data updated, the drive can recover by recomputing the stored digests for all dirty inodes.

- The stored digest can be flagged as dirty and its inode flushed to the media before any data is written out to the media. After the data is successfully written, the inode can be flagged as clean and lazily written to disk. If the drive fails and a dirty digest is read, the drive will know that the digest may be inconsistent with the data and that it should recalculate the digest. This does not require any additional hardware support but it does require synchronous updates to metadata which could cause poor scheduling of the disk head.
- Optimistically, the drive assumes that the system will not fail and all the updates will eventually be flushed to the media. If a failure does occur, the client will detect that the MAC was incorrect when an out-of-date digest is used to generate a MAC of a request. The client will notify the drive to resend the data which, in this approach, will force recalculation of the digests and a resend if the digest was mismatched. If the recalculated digest is the same as the stored digest, then the original reply to the client had been tampered with and a security violation should be logged. Alternately, the drive either explicitly updates/verifies all the digests when crash-recovery occurs or flags all stored digests as invalid during crash recovery.

Of the three, NVRAM is the most powerful because it allows quick recovery without significantly hampering drive performance by requiring synchronous metadata updates sequences. However, the NVRAM approach requires additional hardware support. The optimistic approach must either pay to recompute all the digests or involve the client in noticing a mismatched digest. If the client is involved, the client and the drive will pay the performance penalty for a mismatched digest at request time rather than the drive paying at restart time.

6.4.4 Hash and MAC for Attributes

Filesystem attributes can benefit from a similar precompute optimization. In the prototype, each object has exported metadata, called the attributes, which contain file-system information as well as a set of common fields that are likely to be used in any filesystem (e.g., size, create time, and modification times). The filesystem-specific data fields are fairly large (256 bytes) in order to provide flexibility to the applications; and so are expensive to MAC on every request. As with object data, the attributes change less frequently than they are accessed. The CMU AFS ‘99 workload shows a ratio of 22:1 between attribute retrieving operations and attribute modifying operations. The Auspex NFS workload shows a similar ratio of 6:1.

The relatively large size of NASD attributes coupled with their static nature makes them suitable for the same “Hash and MAC” optimization that I presented earlier. By storing a precomputed message digest along with each NASD attribute, we reduce the cost of protecting the integrity of the 336 byte NASD attribute to calculating a MAC on its representative 20-byte digest.

I can apply this optimization to NASD attributes because, unlike traditional UNIX attributes, NASD does not maintain a last access time. If NASD maintained a last access time then the attribute would change on every operation and storing a precomputed digest with each attribute would not be advantageous because the digests would need to be updated on every request.

6.5 Efficient support for small operations with stored digests

Small requests are an important class of operations in a storage system. NASD needs to support a wide range of workloads from filesystems to databases, with a wide range of access patterns. If we examine current systems, we see a diverse set of access patterns. For example, in an academic research / software development group, typical file system accesses are small while most data is moved in large requests [Baker91, Riedel96]. In addition to file systems, databases and persistent object systems operate on small objects [Stamos84]. These results tell us that both small access and large transfers are important to consider for NASD. Section 6.4 discussed support for large read accesses using stored message digest, but small accesses pose a different set of issues. Optimizing small accesses is the focus of this section.

In the NASD prototype results presented in Section 6.1, the networking stacks introduce substantial per-request overhead, making it difficult to achieve high performance when using small requests and thus minimizing the impact of cryptography on performance as shown in Figure 6-6. Both UNET [vonEicken95] and the VIA [Intel97] are harbingers of commodity networks that provide high throughput and low latency with relatively small transfer sizes. Part of their approach is to provide greater functionality in the network interface thus simplifying the application protocol stack and avoiding operating system intervention. GigaNet has demonstrated a native VIA implementation in their GNN 1000 adapter card, which provides 1.25 Gbps with 8 microseconds latency and under 10% CPU utilization [Giganet98]. Their demonstration system achieve about 80% of its peak bandwidth with 1KB transfers. This is significantly better than the 64KB transfers necessary for the prototype to achieve 80% of its peak bandwidth, which is still smaller than the underlying network bandwidth.

With VIA delivering high bandwidth for small requests, cryptography's impact on performance for small requests will become more pronounced as long as the cryptographic performance and peak drive performance are less than network performance. When maintaining accurate stored digests, maintenance of the stored digests for small updates may result in a substantial loss of throughput which will be more evident with VIA-like technologies.

One approach to handling small writes is to defer the updates of the stored digest and perhaps amortize the update cost across multiple small writes. If only a single write to the disk block occurs, the stored digest update is simply postponed from the write operation to a subsequent read operation. Deferring the update of the stored digest does not improve performance for small read operations and will not help when sequential small writes to the same object are unlikely. Finally, deferring the update of the stored

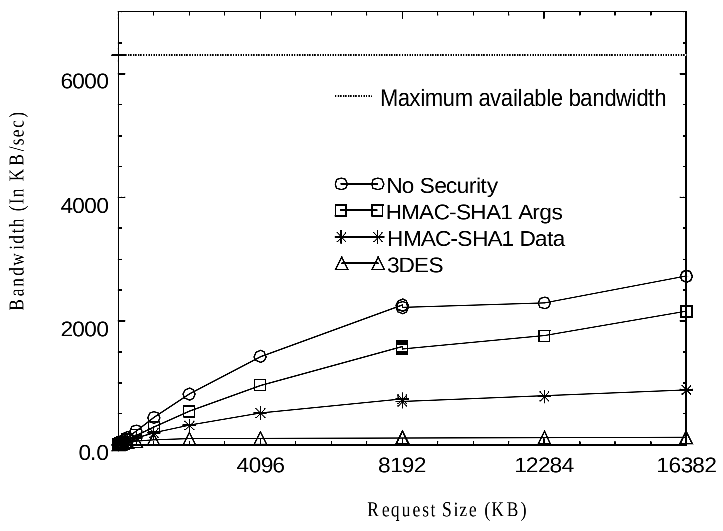


Figure 6-6 Software Cryptography with Small Requests

With small requests, the underlying prototype with and without security is unable to approach its peak bandwidth. The x-axis shows the size of requests, which start at the beginning of an object, made by the client. The y-axis marks the average read bandwidth. Each point represents read throughput for a minimum of 3 seconds of continuous requests and a minimum of 100 requests by a single client reading data from an in-memory object at the drive.

digest to the next read request will create unpredictable performance from the clients perspective. As long as the stored digest is up to date, the client can make reasonable assumptions about expected request service time based on the request size, request alignment, and overall application state. If the digest may need to be recomputed, this introduces another variable that the client cannot predict because it depends on the history of a given disk block.

Even with up-to-date digests, the drive may not always need to recalculate a stored digest for a small write. If the small write starts on a disk block boundary and extends or overwrite the final partial disk block, then the drive will calculate the digest for the final partial disk block when it verifies the data received from the client. This allows the drive to simply overwrite the stored digest rather than recalculate the stored digest for the entire partially-used disk block. In our PDL AFS ‘99 traces, AFS shows this extending or overwriting behavior because AFS operates on 64 KB blocks when updating the middle of a file. In contrast, the Berkeley NFS traces shows a substantial number of operations that are not aligned on 8K boundaries as shown in Table 6-1 which would require a stored digest to be updated. This demonstrates that some systems that have already been “adapted” to NASD would benefit from efficient support for integrity on small requests and non-disk-block aligned requests while still providing the benefits of stored digests to the larger aligned requests. Other systems, yet to be adapted to NASD, may also be able to take advantage of efficient support for integrity on small requests.

6.5.1 Incremental Hashing

The incremental hashing paradigm developed by Bellare et al. [Bellare94, Bellare97b] describes a class of hash functions for which the work to update a previously computed digest is proportional to the size of the change. For network attached storage, this enables small writes and offset writes to be implemented more efficiently.

I will briefly describe the incremental hashing paradigm and specific examples of incremental hash functions described in [Bellare97b] and then describe the specific instantiation of the function that I propose for network attached storage.

Incremental hashing takes a message and divides it into a sequence $x_1, x_2, \ldots, x_m$ of fixed sized blocks of size *b*, called incremental blocks. This is the basic granularity of change for a hash function. If a two- byte change spans two incremental blocks, it is twice as expensive as a two byte change that falls within a single incremental block. Each incremental block is concatenated with its block number to generate an *augmented* block, $x_i' = i.x_i$. For each augmented block x_i', apply a compression function *h* to x_i' to get the hash value $y_i = h(x_i')$. Combine $y_1, y_2, \ldots, y_m$ using a combining operator (∇) to get a final hash value.

More clearly we can express this as follows:

$$HASH(x_1, \ldots, x_n) = \nabla_{i = 0, n} h(i.x_i)$$

To replace an incremental block x_i with a new incremental block $x_i^{\dagger}$ in a stored digest, we must compute $h(i.x_i)$ and take an inverse of the stored hash and then combine in $h(i.x_i^{\dagger})$ into the stored hash. This requires less work than recalculating the entire stored digest.

Table 6-1 Berkeley NFS Request Alignment

In this workload, there is a significant amount of variety in the alignment of client requests. While most requests are aligned on 8K boundaries, almost 1/3 of the write requests are not aligned and a small percentage of read requests are not aligned.

Operation/Alignment	8K	7K	6K	5K	4K	3K	2K	1K	Other
Read	97.3	0.0	0.0	0.0	2.6	0.0	0.0	0.0	0.1
Write	67.8	1.2	1.2	1.2	25.2	1.1	1.1	1.1	0.1

Another property of the incremental digest is the ability to compute the digest of part of a disk block without computing over all the data being read. Observe that

$$HASH(x_i, \ldots, x_j) = \langle \nabla_{i = 0, n} h(i.x_i) \rangle \nabla^{-1} \langle \nabla_{i = 0 \ldots i-1, j \ldots n} h(i.x_i) \rangle = \nabla_{i = i, j} h(i.x_i)$$

In English, if we need the hash of only part of the data covered by a stored digest, the desired hash can be computed by taking the inverse of the combination of the stored hash and the hash of the complement of the portion being requested or computed directly from the data being requested. Thus, by using this complementary property, we will need to compute over, at most, half the data to calculate a correct hash for any portion of the data.

Bellare presents two classes of incremental hash functions: MuHash and AdHash where the combining operators are modular multiplication and modular addition, respectively. The security of MuHash is reduced to the difficulty of the discrete logarithm problem while the security of AdHash is reduced to the difficulty of the subset sum problem.

For use in NASD, AdHash is more appealing than MuHash for two reasons: the size of digests and computational cost. Observe that the size of the modulos is equivalent to the size of the digest we must store with each disk block. For MuHASH, the modulos must be at least 512 to 1024 bits in order for discrete logarithm problem to be difficulty. For AdHASH, current approaches can usually solve random subset sum problems of 100 bits, but 200 bits is well beyond their reach; however, Daniele Micciancio, one of the researchers working on incremental cryptography, recommends at least 256 bits for longer term security [Micciancio99]. This provides a factor of 2 to 4 difference in space required to store the digests with each disk block in NASD. Secondly, addition is faster than multiplication so AdHash will have better performance in both software and hardware implementations.

6.5.2 Incremental Digests for NASD

A NASD system could use AdHASH built on the SHA-1 compression function and addition a modulo 2^{256}. Ideally, I would like to simply use the SHA-1 compression function to produce 160-bit hash values that would be combined through addition modulo 2^{160}. Unfortunately, 160-bit hash values would produce subset sum problems that are almost solvable with current techniques. Instead, I generate a 256-bit hash by applying the SHA-1 compression function to two sequential message blocks. Applying the compression function twice, $c_i = compress(x_i)$ and $c_{i+1} = compress(x_{i+1})$, produces two 160-bit digests which are combined by shifting the first one left 96 bytes and XORing the values together to produce a 256-bit hash:

$$h(x_i x_{i+1}) = (compress(x_i) \ll 96) \oplus compress(x_{i+1})$$

This approach provides all the collision resistance of the original compression function because an adversary must find a collision on all the output bits in order for a collision to be useful and the hash function produces a 256-bit result which makes the subset sum problem difficult. An obvious alternative is to simply concatenate the results of the two calls to the compression function and generate a 320-bit output. However, a 320-bit digest requires the drive store 320 bits per disk block, which is a larger overhead than storing only 256 bits. If subset sum attacks improve significantly, this approach can easily be adapted to produce 320-bit subset sum problems.

Storing 256-bit digests rather than 160-bit digests used in HierMAC further decreases the addressability of NASD inodes. A single inode can only address 1.3 MB of data compared to 1.8 MB with HierMAC. However, a different allocation of inode space between direct and indirect pointers can yield significantly different capacity and indirection levels in the drive filesystem.

The incremental hash functions described by Bellare et al. concatenate the block number *i* onto data block x_i before hashing in order to prevent reordering of data blocks. This can double the number of invocations of the compression function. If the incremental blocks are the same size as the basic block of the compression function, which allows fine grain changes to be done most efficiently, then we need to invoke the compression function twice, once for the data and once for the block ID, which doubles the amount of hashing. The obvious solution is to increase the incremental block size to amortize the cost of appending the block number. However, increasing the incremental block size reduces our potential gains from using incremental cryptography because all updates of a stored digest occur on the granularity of an incremental block.

I propose two potential solutions to this problem for NASD:

1. Incorporating the block number into the initial vector of the compression function or
2. Multiplication of the hash of the incremental block by 3^i (or any value relatively prime to the modulos).

In an iterated hash function, shown in Figure 6-7 and of which SHA-1 is an example, the initial vector (IV) is used as an initial seed value for the first iteration and a chaining variable between iterations. By placing a value in the IV, the final result of the hash function is dependent on the value. If this were not true, the final output of an iterated hash function would not be dependent on the results of previous iterations.

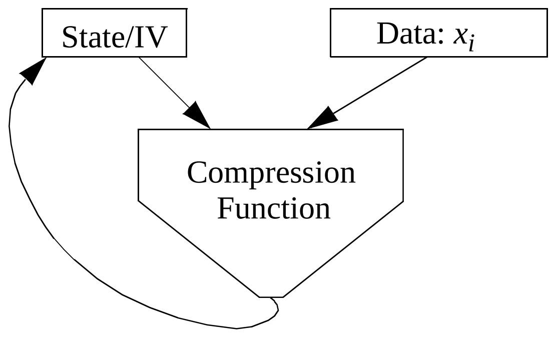


Figure 6-7 Structure of an Iterated Hash Function

An iterated hash function, such as SHA-1 or MD-5, is built around a compression function that takes a state variable and a fixed size data block and produces a new state variable. The message is first padded out to an integral number of data blocks and then fed into the compression function. Initially, the state variable is set to a predefined valued called the IV. The compression function updates the state variable after each message block is processed. The final digest is normally a simple function of the state variable.

Does changing the IV make it easier to find a collision? Although SHA-1's design criteria are not public, I believe it unlikely that changing the IV will make collisions more likely. SHA-1's IV is a simple sequence of bytes with a very regular pattern and thus provides little evidence of being a particularly special value. Additionally, SHA-1 is directly derived from MD4 [Rivest91], which was developed openly and has no publicly stated special design criterion for the IV. In the initial MD4 paper, Rivest suggests changing the IV, along with the other constants, and running two-MD4 functions in parallel to generate longer digest values which indicates some flexibility in the IV value. Preneel and Oorschot also modify the IV of arbitrary hash functions in their MDx-MAC construction to build a message authentication code [Preneel95]. Together, these facts make it unlikely that there is something special about the values employed in SHA-1's IV and it should be safe to effectively incorporate the block ID into the message digest using the IV. Furthermore, this technique is applicable to any hash function that does not use special values in the IV.

A second potential solution would be to multiply the digests of incremental blocks by 3^i to provide the ordering property. For each incremental block, this solution adds the cost of an exponentiation, which we could tabulate a priori, and a modular multiplication of very long integers to the cost of the compression function. For a compression function where changing the IV is a concern, this solution is more appropriate.

6.5.3 Comparison of Cryptographic Cost

In this section, I analyze the amount of message digest or MAC computation required to provide integrity for clients accessing network attached storage devices. This analysis is a first-order approximation of the impact of providing integrity on drive throughput.

I compare three approaches:

Basic MAC: The basic MAC scheme where all bytes are MAC'd using HMAC-SHA1.

Stored Digest: The "Hash and MAC" approach using precomputed SHA-1 digests stored with disk blocks as described in Section 6.4.

Incremental Stored Digest: The "Hash and MAC" approach using precomputed digests that are generated using AdHash built on SHA-1 and addition modulo 2^{256} with the block number placed in the IV. The precomputed digests are bound to a key using HMAC-SHA1.

The comparison is in terms of the number of invocations of the SHA-1 compression function which is the "common currency" of cost in the three approaches. The cost of padding out the messages to message digests or message authentication code block sizes is assumed to be zero. This is the cost of filling a buffer with up to 64 bytes of zeros and perhaps the message length which is a cheap operation relative to the compression function calls and will only occur once or twice per request. In contrast, the modular addition and subtraction used in the incremental stored digest approach will be used many times in a single request so must be modeled more accurately. The cost of modular addition and subtraction are modeled as 0.10 and 0.07 the cost of an invocation of the SHA-1 compression function. These values are the relative execution cycle counts, measured using DEC's ATOM profiling tools on a 233 MHz Alpha 21064, of simple C-language compiler-optimized implementations of 256 bit addition and subtraction using only 32 bit variables compared to our SHA-1 compression function implementation. The costs are likely higher than a hand-optimized assembly implementation but provide a conservative estimate of the cost of the combine and uncombine operations. These costs, as well as equations describing the costs of all three approaches, were modeled in Mathematica to generate the data presented in Section 6.5.

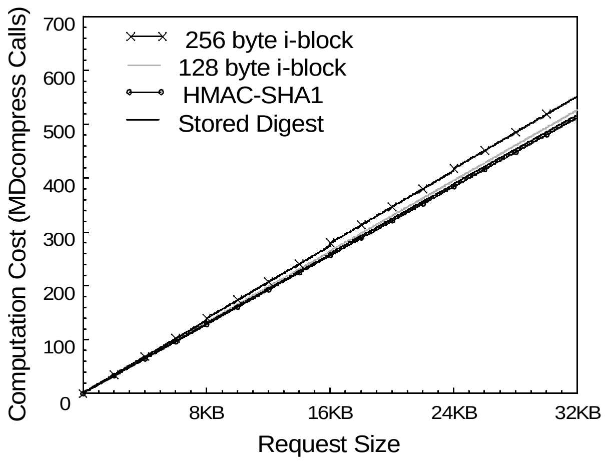


Figure 6-8 Client Overhead for Integrity

Using the precomputed digest optimizations requires clients to do a small amount of extra work for each disk block transferred. Based on a model of what computation the clients must perform, each line shows how many times a client must call the compression function, or equivalent work in the combine and uncombine operators, for a given amount of data being transferred. The x-axis is the size of the request and the y-axis is the number of invocations of the SHA-1 compression function, the core of SHA-1. The first two lines are approximations of the cost when using incremental cryptography on 256 byte and 128 byte incremental blocks respectively. HMAC-SHA1 is both the baseline for comparison and the minimal achievable amount of computation. Stored digest is the cost for the "Hash and MAC" approach with SHA-1 as described in Section 6.4.

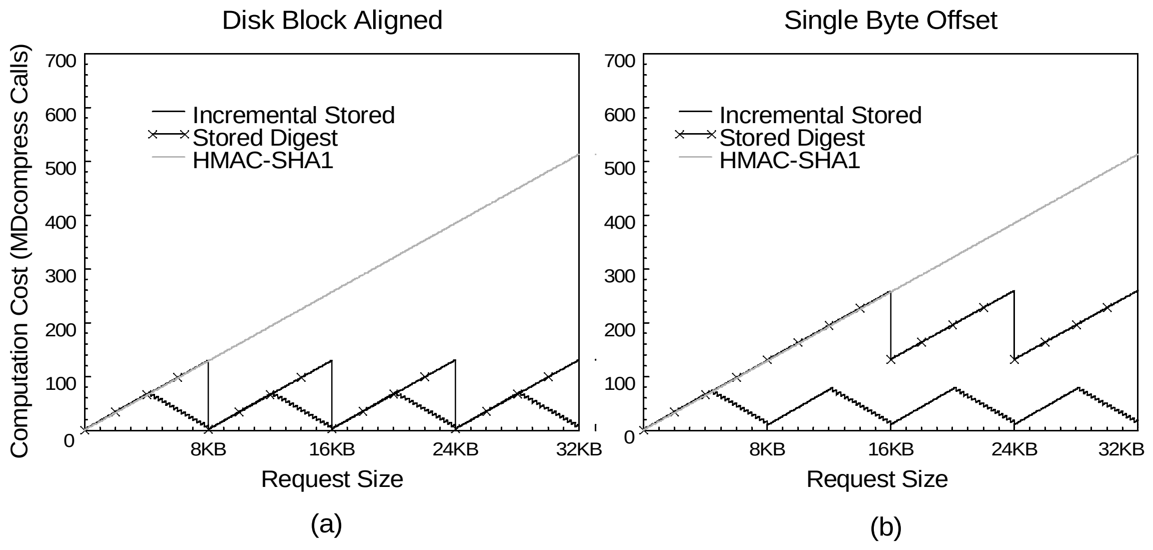
6.5.3.1 Integrity Overhead Costs at the Client

As shown in Figure 6-8, both of the stored digest solutions add a small amount of overhead to clients. This overhead is paid when both sending and receiving data so the curves are independent of a request being a read or a write. While the drive reduces its work by using the precomputed optimizations, the client performs a small amount of extra work to MAC the message digests. In following sections, I will explore the advantage to the drive of using incremental stored digests but, in this section, I examine the cost borne by the client. The basic SHA-1 has the minimal amount of cryptography necessary to provide integrity. Using the Hash and MAC structures adds a very small overhead to perform the MAC of the concatenated message digest.

The incremental approach adds computational overhead because the combine operator, modular addition, is applied to generate a single digest for an entire disk block from the results of the compression function on each incremental block. If we use the smallest possible block size as the basis for generating incremental digests (e.g. 128 bytes, the basic size for the 256 bit digests described in Section 6.5.2) then the addition of 64 256-bit values per 8K disk block provides a small but noticeable overhead. With hardware support, this overhead could be hidden by pipelining the computation of the digests and

Figure 6-9 Drive Cryptographic Cost for Integrity on Reads

Both incremental-stored and simple-stored approaches significantly reduce the amount of cryptographic work the drive must perform on a large read request compared to HMAC-SHA1. For misaligned reads, the complementary property of incremental digests allows the digests to be calculated more easily than normal digests. Based on a model of what computation the clients must perform, each line shows how many times a client must call the compression function, or equivalent work in the combine and uncombine operators, for a given amount of data being read from a given offsets. The x-axis is the size of the request and the y-axis is the number of invocations of the SHA-1 compression function, the core of SHA-1. Incremental stored is the incremental scheme described in Section 6.5.2 using 256 byte incremental blocks. Stored digest is the cost for the "Hash and MAC" approach with SHA-1 as described in Section 6.4. HMAC-SHA1 is the most standard way of providing communication integrity and is used as a basis for comparison.



the additions. However, simple drives without hardware support and clients will not be able to hide this overhead cost. With larger data transfers, overlapping computation and data transfers should hide much of this overhead. To be conservative, I show 256-byte incremental blocks which requires half the overhead and still allows the drive to reap much of the benefits of incremental digests. I will use 256-byte incremental blocks for the remainder of the evaluation.

6.5.3.1 Integrity Cost at the Drive for Reads

Figure 6-9a shows that both the stored digest and incremental stored digests approaches perform significantly less cryptographic work for large block aligned reads. The basic SHA-1 approach has a constant per-byte cost. The stored digest approach can use the stored value whenever an entire disk is read; thus, performance can improve dramatically, as discussed in Section 6.4.2, for large reads. Stored incremental digests gain similar benefits for large disk block reads and also reduce the cost for some smaller reads because the complementary property of incremental digests allows the drive to reduce its computation to hashing only half a disk block. In a system where small requests were efficient, I expect this reduction to translate into a smoothing out of the sawtooth behavior that was shown in both Figure 6-3 and Figure 6-4.

For non-aligned reads, as shown in Figure 6-9a, the precomputed digests do not provide any benefit until the drive reads almost 2 full disk blocks. The drive must always compute a digest on all the data read in the first partial disk-block because the drive only stores a digest of the entire disk-block. The incremental stored digest has a much smaller penalty because the complementary property of incremental digests is largely independent of the offsets and provides benefit from the precomputed digests even for arbitrarily aligned reads.

Both Figure 6-9a and Figure 6-9b show a small saw-tooth behavior for the incremental-stored digests because of the 256-byte incremental block size. The sawtooth is the result of the extra work required to generate digests for partial incremental blocks. Just as the stored digest approach must calculate on-the-fly a hash of all bytes in a partially read disk block, the incremental stored approach must calculate on-the-fly a hash of all bytes in a partially read incremental block. For smaller incremental block sizes, the tooth size will shrink while larger incremental block sizes will increase the size of the teeth. However, reducing the incremental block size increases the overhead because of more combine operations as was shown in Figure 6-8.

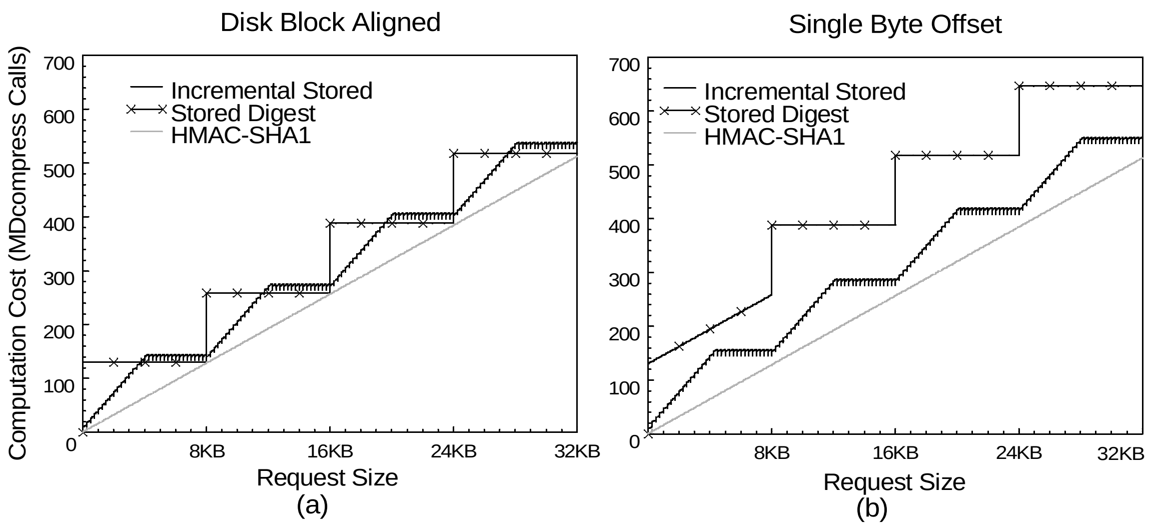
6.5.3.1 Integrity Cost for Writes at the Drive

For writes, incremental digests and stored digests add up to one disks block's worth of work to small or misaligned operations. The drive must verify the received MAC and then update the stored message digest in both the stored and incremental stored approaches. For the stored digest approach, the drive must generate an entire new stored digest for a disk block even if only a single byte is written. This is a substantial penalty for small writes. If a write starts on a disk block boundary then computing the new stored digest can simply continue from the calculation necessary to verify the digest on the data received from the client, since recomputing the hash of the common prefix would be redundant. In this case, the cost is a function of the number of disk blocks touched by the write operation which creates the step-function effect shown in Figure 6-9a.

If a write begins offset into the disk block, shown in Figure 6-10b, the stored digest approach pays a larger penalty than the incremental stored approach. With the basic stored digest scheme, the drive can no longer continue the calculation used to verify the data received from the client because the received data is no longer a prefix of the disk block. Instead, the drive must first verify the received MAC and then start from scratch to generate the stored digest for the updated disk block. This makes small, miss-aligned writes extremely expensive. The incremental approach is largely independent of offset and does not pay these penalties on small writes.

Figure 6-10 Drive Cryptographic Cost for Integrity on Writes

For write operations, both stored digest approaches pay a penalty for updating partially modified disk blocks. For misaligned operations, this penalty is reduced when incremental digests are used. Based on a model of what computation the clients must perform, each line shows how many times a client must call the compression function, or equivalent work in the combine and uncombine operators, for a given amount of data being written to a given offset.The x-axis is the size of the request and the y-axis is the number of invocations of the SHA-1 compression function, the core of SHA-1. Incremental stored is the incremental scheme described in Section 6.5.2 using 256 byte incremental blocks. Stored digest is the cost for the "Hash and MAC" approach with SHA-1 as described in Section 6.4. HMAC-SHA1 is the most standard way of providing communication integrity and is used as a basis for comparison.



6.5.3.1 Incremental Advantage for Clients

Incremental cryptography can sometimes reduce the cryptographic task of the clients. When a client requests a large amount data, such as 64KB chunk in AFS, and modifies only a few bytes, incremental cryptography allows clients to generate updated digests for the entire 64KB chunk without recomputing over 64KB. This avoids the client computing digests of all the data when it is written back to storage. In theory, the application could simply write the changed bytes back to storage rather than the entire 64KB chunk. Frequently, it is more efficient simply to flag the entire chunk as dirty rather than individual byte ranges thus the entire chunk must eventually be flushed to the storage system. The amount of cryptographic work to protect the integrity for the client on a write is a function of the number of bytes changed and their frequency of change rather than the size of the chunk.

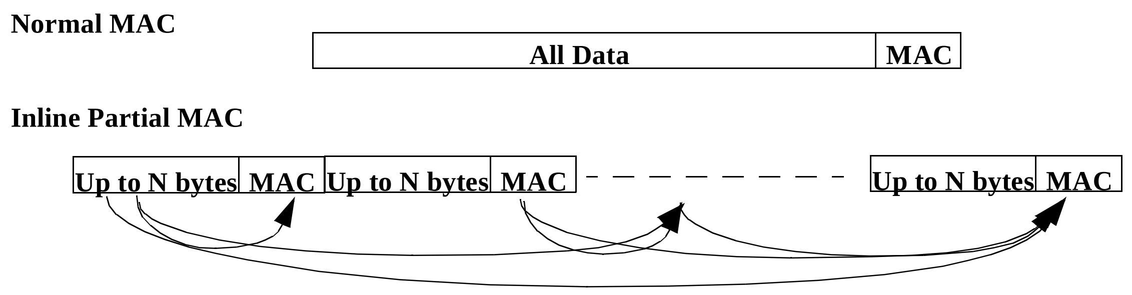


Figure 6-11 Inline Message Authentication Codes

A normal MAC protects "all" of the data. By injecting MACs of all data up to the current point, the receiver only needs to buffer N bytes before it can begin process. All MACs are cumulative of all data up to the current point to preserve the relationship between chunks of data.

6.6 Improving Receiver Buffering

For receivers (e.g. storage on writes, clients on reads), performing on-the-fly cryptographic work is unavoidable. Faster CPUs can improve the situation but networking already consumes up to 80% of the host's cycles in protocol processing overhead, leaving few excess CPU cycles for cryptography [Gibson98]. Worse, to increase network transfer efficiency, applications often optimize for large transfers. With security added to the transfer, it is possible that a receiver would have to buffer an entire message before it could verify the data's integrity. This can be a problem for storage servers or NASDs, where limited (and shared) resources make it difficult to dedicate large amounts of buffers. Further, delaying verification until all the data is received prohibits pipelining of data processing (e.g., writing to storage) upon the reception of the data.

In order to reduce the buffering requirement at the receiver and permit pipelining of data as it is received, we insert digests of prefixes of the requests into the data steam as illustrated in Figure 6-11. At some regular interval, the sender inserts a MAC of all the data sent so far. When the receiver receives a MAC, the receiver can verify that *all data up to the MAC* is valid and then begin processing the data. This slightly changes the semantics of the MAC, from all-or-nothing to having prefixes be independently valid. Effectively, we are treating a large data transfer as a stream of data rather than a block of data. The primary advantage is to reduce the buffering requirement at the receiver from the full request down to the amount of data sent between MACs. This is necessary for providing security over any large data transfers and is applicable regardless of what type of message authentication code you are using. For NASDs, with their resource-constrained environment, it is very important to process data as it arrives rather than buffering an entire write request because a single write may overflow all of the drive's available memory.

6.7 Related Work

The “Hash and MAC” approach borrows its basic idea from the web server community. Kaashoek et al. demonstrated that precomputation of TCP/IP checksums on web pages can help increase web server throughput but a factor of 2.3 [Kaashoek96]. The stored hash values in NASD exploit workload similarities between web servers and fileservers. However, the “Hash and MAC” construction also handles the following concerns: any stored work must be usable for multiple clients, a MAC must be bound to a key, and clients can make requests on arbitrary block boundaries.

“Hash and MAC” is constructed out of two cryptographic building blocks: a MAC and a hash function. For a good list of modern MACs and hash functions, the reader should consult [Menezes98]. For the NASD experiments, I have specifically instantiated the hash function with SHA-1 [FIPS180-1] and the MAC with HMAC-SHA1 [Bellare96a] which are both widely used cryptographic functions.

The basic structure of “Hash and MAC” is similar to the more traditional “Hash and Sign” approach used with public key cryptography. Both approaches admit a set of offline attacks that are not possible with a normal MAC. “Hash and MAC” will be faster because it uses a MAC rather than public key cryptography to sign the hash. Additionally, “Hash and MAC” does not rely on any special number-theoretic properties of the key so it can be used with any key distribution mechanism while “Hash and Sign” relies on certain number theoretic properties of the public and private keys.

In order to deliver the advantages of “Hash and MAC” on small or misaligned requests, I extended it with an incremental hash function, specifically AdHash [Bellare97b]. Since the motivation for using incremental hash functions is to avoid excess cryptographic work and make small operations more efficient, NASD can’t use the concatenation of block identifiers to the data blocks to prevent reordering which is the original design of AdHash. Instead, NASD binds the block IDs into the hash by using the IV of the hash function or multiplication by 3^i, presented in Section 6.5.2. Additionally, because the security of AdHash depends on the difficulty of the subset sum problem, the 160 bit hash values generated by SHA-1 are on the edge of breakable. As a result, NASD combines two SHA-1 digests by a combination of shift and XOR to produce a 256 bit hash.

The UMAC construction takes another approach to building a fast and secure message authentication code [Black99]. UMAC builds a new universal hash-function family which can exploit the SIMD parallelism of modern processor architectures, such as the Intel Pentium with MMX. Black et al. have shown how to embed a universal hash-function into a MAC in a formal manner. In contrast, NASD’s HierMAC builds on existing widely used hash functions which use heuristic, rather than provable security, to justify its strength. However, HierMAC is based on well established primitives that have withstood years of attack by the security community.

In some systems, the added performance of UMAC may be sufficient to meet the drive's throughput requirements. Even when UMAC is too slow to meet NASD's requirements, UMAC is faster than SHA-1 and can be used to replace HMAC-SHA1 in the HierMAC design. While it is possible to find MACs that are faster than HMAC-SHA1, and weaker, UMAC has the advantage of being both faster and having a strong theoretical basis for security. Using HierMAC with UMAC in NASD will further reduce the request-time computational requirements from HierMAC with HMAC-SHA1.

6.8 Discussion & Conclusion

This chapter has presented two techniques to reduce the amount of computation required when storage provides communication integrity. This enables low-cost drives, or drives under provisioned with security resources, to provide high bandwidth integrity-protected communication. By taking advantage of the fact that data is read more frequently than it is updated, I have shown that storage can use *stored* security information to protect integrity and increase the integrity-protected bandwidth available from a prototype drive. While this does not improve the task for providing privacy, it is *critical* for storage to provide integrity guarantees because nobody else in the system can provide this property, while privacy can potentially be handled at the application layer. Furthermore, I have shown that by applying techniques of incremental hashing we can improve the performance for misaligned and small read and write operations.

Using stored information is not without a price. I have opened the door to a more convenient attack by off-line computation which does not require an oracle. With a good message digest function, this risk is small but it is an *additional* avenue of attack. In an ideal resource-rich environment, a drive would have ample resources for all its tasks and the trade-off would be unnecessary. Unfortunately, real-world constraints motivate drives to be very cost conscious devices, so some trade-off decisions will be made in any design.

For misaligned and small writes, the stored digest will also add, on average, a disks block worth of hash processing to the operation to keep the stored digests current. If the frequency of retrieving data significantly exceeds the frequency of updates, we are reducing the total amount of cryptographic computation necessary and the expected cryptography per operation. When the strength of the message digest is a concern, the difficulty of the additional attacks should raised by adopting a longer and stronger message digest. While a stronger digest may be more expensive to calculate, if the disparity between retrievals and updates of data is great enough then we will still have a performance win.

By incorporating incremental cryptography, I have reduced the penalties for small or misaligned operations that are associated with a stored digest at the price of a small increase in per-byte overhead. From the security perspective, NASD using incremental cryptography can be broken if an adversary can solve a random subset sum problem, with the specific structure that NASD produces. Using current state-of-the-art approaches, solving a random subset sum problem is computationally difficult. However, if a new algorithm were found to solve arbitrary subset sum problems, such an algorithm could be used to attack a NASD system built on incremental cryptography.

Chapter 7: Hardware for Security Performance

Cryptography capable of sustaining network data rates is the ideal solution for any storage workload that requires security. However, cost considerations can make this difficult to achieve. Alternatives, such as the HierMAC or HierMAC with incremental digests (Chapter 6), reduce the amount of cryptography required per byte for read traffic, but do not significantly improve small transfers or write-traffic bandwidth — both essential to achieving acceptable storage performance. Fortunately, acceptable storage performance at sub-network cryptographic speeds is possible because: 1) media data rates are significantly lower than high-speed network data rates; and 2) storage workloads have periods of idleness. These characteristics provide a range of performance, between network and media data rates, that enables an unique set of trade-offs involving cost, throughput, and latency.

The decision to protect integrity, privacy, or both affects how security creates latency. Integrity and privacy are built on different cryptographic primitives, message authentication codes and encryption respectively, which process data in differently sized blocks. More importantly, privacy is more like a transformation of a continuous stream of data while integrity requires the receiver to decide, at discrete intervals, if the data has been modified on the network. This fundamentally introduces latency as the receiver awaits enough data to determine if a request was modified.

This chapter explores how this performance range can be exploited to achieve good system performance without implementing full network-speed cryptography. To ground the discussion, I begin with an overview of drive electronics and examine the performance of current software and hardware solutions. Next, I show how integrating security into a NASD impacts the system's latency and discuss the performance issues involved in providing integrity, privacy, or both. The analysis is quantified using real file system traces and reveals that a drive, using HierMAC and providing only 33% of a network's full-duplex bandwidth, can successfully services file system requests with less than a 10% increase in latency (over a system with no security). Finally, I discuss available hardware solutions to integrate cryptography into a drive's central ASIC as well as the potential advantages of reconfigurable computing.

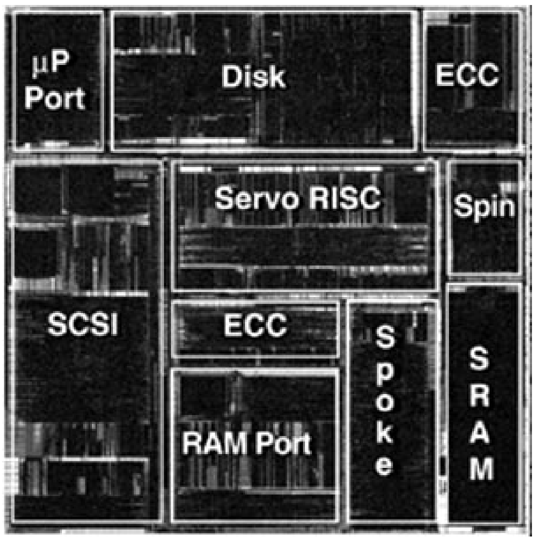


Figure 7-1 Quantum Trident ASIC

Modern drive ASICs integrate a large amount of functionality onto a single chip. The SCSI controller, servo controller, sequencer, motor controller, error correcting code, and a small amount of SRAM provide the core device functionality while a CPU and DRAM are on other chips. The primary ASIC in the Trident consumes approximately 110 thousand gates and 22 Kb of SRAM in a 74 sq. mm package using 0.68 micron chip technology.

7.1 The Physical Architecture of a Network Attached Secure Disk?

The electronics on a modern SCSI disk drive are very similar to a modern computer, and include a microprocessor (~60 MIPS), a SCSI interface, and several megabytes of RAM. In addition, there are numerous very small functional units that manage the drive, including a motor controller, R/W channel, preamp and write driver, error correcting code engine, sequencer, buffer controller, servo controller. While previous generations spread these functional units across multiple chips, increasing VLSI integration levels are reducing the number of physical chips — thus reducing failure rates, decreasing cost, and increasing performance. For example, a modern Quantum Viking disk drive integrates the SCSI controller, sequencer, error correcting code engine, motor controller, and servo controller onto a single ASIC, the Trident (Figure 7-1), while the memory, microprocessor, DRAM, R/W channel, preamp & write driver, and motor controller are in separate chips. Further, recent announcements from the drive industry suggest that single-chip solutions are on the horizon [Lammers99].

A network attached disk would require the same core function as a SCSI drive, replacing the physical SCSI interface with a high-performance network (e.g., Gigabit Ethernet, FibreChannel) while increasing the microprocessor's performance to support NASD's in-drive file system. Adding hardware-based security requires four new functional blocks: key memory, encryption/decryption, message authentication code (which uses SHA-1 in the prototype), and key management logic. Software-based security would require fewer functional blocks, with the encryption/decryption and key management blocks handled by the microprocessor and keys stored on the disk media. However, software-based security demands a significantly more powerful processor. Further, because the microprocessor must touch all bytes that are either sent or received, the ASIC's internal datapath, which is currently optimized for minimal data movement through the microprocessor, would also require a fundamental change.

7.2 Software Cryptography: A Performance Bottleneck

Most cryptographic algorithms are not designed with efficient software implementation as a primary design criterion. However, security's increasing importance has fostered much greater interest in cryptographic algorithms that can be efficiently implemented in software. For example, the Fast Software Encryption International Workshop is a testament to the importance of high-performance software cryptography. Software performance is now one of the main criterion in designing future standard encryption algorithms [Schneier99, NIST98] and the characteristics that make an encryption algorithm fast on an Intel Pentium family of processors are seriously being explored [Schneier97].

Current workhorse encryption algorithms require significant computational power. For example, Triple-DES[1] (also called 3DES), which predates the popular interest in security, requires 108 clock cycles per byte on a Pentium processor [Schneier97]. This places 3DES's maximum throughput for a GHz Pentium processor at only ~9MBytes/second.

The likely successors to Triple-DES, the Advanced Encryption Standard (AES) candidates [NIST98], all improve on the performance of Triple-DES but still require 20-69 clock cycles per byte for 8 KB requests with an average penalty of an additional three cycles per byte on smaller, 1 KB requests [Schneier99]. Assuming the drive processor is a 200 MHz Pentium processor, the most promising of the AES candidates will only deliver 10 MB/sec, well below the media rates of current disk drives and completely insufficient for future disk drives.

Hash functions have significantly better software performance than encryption. For example, SHA-1 on a Pentium requires 13 clock cycles per byte (15 MB/second) while RIPE-MD160, another strong hash function, hashes at 16 clock cycles per byte (12.5MB/second) [Preneel98]. While better than the fastest AES algorithms, they will still consume most of a 200 MHz Pentium's cycles supporting the media rates of current disk drives.

These performance numbers show that a 200 MIP processor (e.g. 200 MHz StrongARM), the class of processors expected on early Network Attached Secure Disks [Gibson98], will be unable to support software-based cryptography. On a 200 MHz StrongARM system, I measured 25 cycles per byte for SHA-1 and 250 cycles per byte for Triple-DES using only compiler optimized C-language code. Assuming a factor of two or three improvement for hand-coded assembly-language optimizations, the cycles per byte for a StrongARM are comparable to the published cycles per bytes for Pentium implementations and are reasonably indicative of what we can expect from an optimized software implementation on a StrongARM. Of course, the details of the processor can

1. Triple-DES was recently proposed as a revised U.S. government Data Encryption Standard (FIPS 46-3), replacing single DES, so we can expect it to be relevant for many years [NIST99].

have a significant impact on the overall performance due to the presence or lack of specific instructions that the cryptographic algorithms heavily utilize, but the literature and my experiments show that it is unlikely that a 200 MIP processor will be able to provide the necessary performance of a drive using only software cryptography.

7.3 Cryptographic Hardware: An Overview

For many years, researchers and industry have built application specific integrated chips (ASIC) that implement many of the basic cryptographic functions used in modern protocols. Eberle at Digital's System Research Center demonstrated an experimental DES chip in 1992 that delivered 1 Gb/s performance [Eberle92]. Currently, you can purchase chips such as the Hi/Fn 7751 [HiFn99] or VLSI's VMS115 [VLSI99] running at 80 MHz which deliver approximately 100 Mb/s and 200 Mb/s performance for both SHA-1 and Triple-DES. These chips, primarily designed to enable IPsec-based virtual private networks in 100Mb/second routers, may not be priced aggressively for commodity devices. Pijinburg Custom Chips' next generation ASIC (500k gates, 0.18 micron) will implement SHA-1, Triple-DES, Safer SK64, and RIPEMD-160 [vanPelt99] and is expected to deliver up to 500 Mb/s performance from each functional unit. Cognitive Designs next generation ASIC, the CDI 3000, will perform Triple-DES at 172 Mb/s and concurrent SHA-1 at 204 Mb/s, priced at $20 in lots of 1,000 [Finley99]. While these cost and performance numbers are difficult to map directly NASD, they do provide an intuition of the performance and cost of readily available hardware support.

7.4 Integrating Security Hardware in Storage Devices

7.4.1 Security and the Drive Datapath

The purpose of adding cryptographic hardware to a storage device is to reduce the latency and increase throughput over software-only solutions. Latency is important because additional latency will translate into increased request service times seen by the client, which clients are sensitive to on small requests, and increased internal buffering requirements within the drive as larger queues are required. Similarly, throughput is important because if the drive has insufficient cryptographic throughput, it will be unable to deliver some of its raw bandwidth to clients. This implies an inefficient utilization of drive resources and that more NASDs to deliver the same aggregate bandwidth to a set of clients are needed.

At a functional level, security adds another stage to the processing of a request that can throttle system throughput and increase latency seen by clients (Figure 7-2). Without security, requests arrive on the drive's network interface. Then they are processed by various levels of communication protocols. Next, they are placed on a work queue where

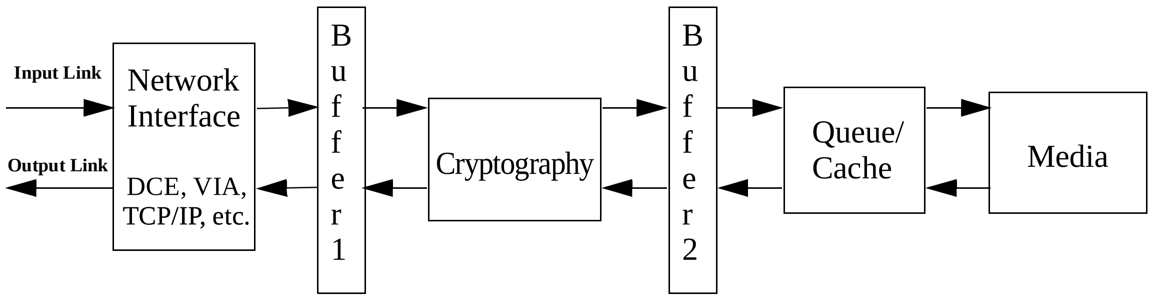


Figure 7-2 Model of a NASD's Internal Functional Pipeline

When security is introduced into a disk drive, the drive may need to buffer requests both before or after the cryptography in order to maintain correctness or perform speed-matching. The Queue/Cache holds requests queued up at the media and the drive's data cache. The buffers and the media queue/cache may allocated from a single memory pool and illustrate a logical distinction rather than a physical one. If the network runs faster than the security but security is faster than the media, buffer1 will fill on a write and buffer 2 on a cache read but both buffers will empty faster than media can drain the queue. If the security is slower than the media rates, buffer1 will fill on every write and buffer2 on every read.

they are either serviced from the cache or sit until they are scheduled for the media access. If the cryptography is slower than the network data rates, data will slowly queue up between the network and crypto on incoming traffic and between the crypto and the drive electronics on the outgoing traffic. However, the outgoing path will be more limited by media data rates (except for cache hits). Even the incoming path is ultimately limited by the media rates since the drive can only buffer a limited amount of data before clients must slow down as data is written out to the media.

7.4.2 Latency

Cryptographic operations impose several ordering dependencies that impact latency. This section examines how these various dependencies and different encryption algorithms influence overall latency. The analysis is based on a simple model that extracts the maximum amount of parallelism available, both in terms of key management and data processing. In reality, real performance will differ, but this model is designed to provide a basic understanding to how various security components benefit from hardware support.

Figure 7-3 shows the chain of dependencies that must be satisfied for the drive to service a request (Figure 7-3a) or send a reply (Figure 7-3b). In both directions, the first step is to determine what key should be used to process the request. On an incoming request, the request header provides enough information for the drive to find the necessary keys in a local cache or, for a key-cache miss, generate the necessary keys. If we miss, we need to generate the access credential key, the MAC of the public portion of the access credential, and then digest the credential key and a pair of constants to generate the actual request-MAC and request-encryption keys (see Section 4.3.2.2 for a more detailed

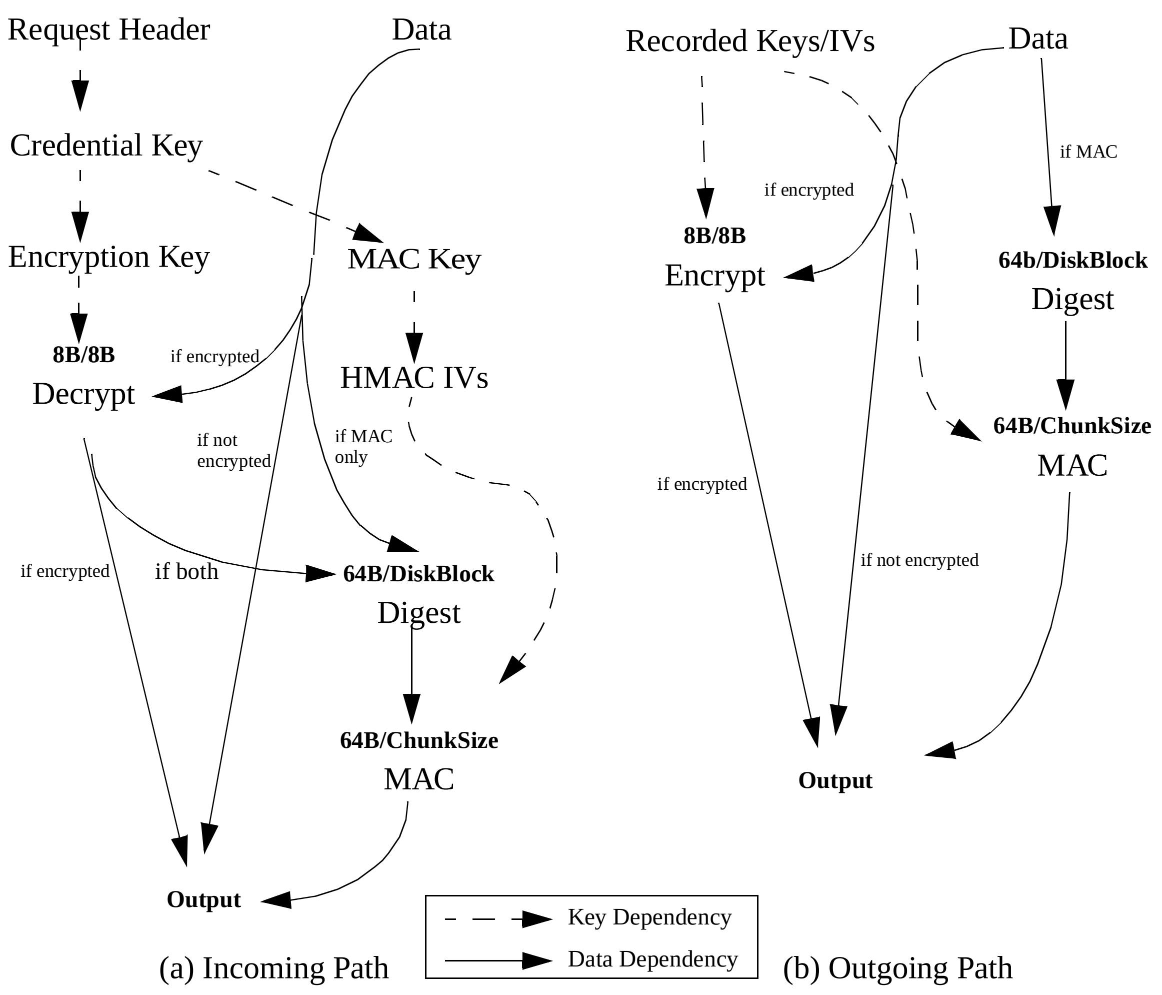


Figure 7-3 Dependency Graph for Security Processing

This graph illustrates the dependencies that add latency to the processing of incoming and outgoing portions of a request for the different phases of the processing involving cryptography. The cryptographic functional blocks are labeled with two values: the number of bytes necessary to start computing and the number of bytes before a result is generated. When HMAC-SHA1 is used, the digest phase will be a null operation and simply pass values directly onto the MAC. When HierMAC is used, digests of disk blocks will be generated in parallel and the results fed into the MAC.

explanation of the generation of the request-MAC and request-encryption keys). The key generation can take a significant amount of time, making the alternative access credentials presented in Chapter 5 an attractive alternative when a drive lacks the hardware necessary to speed key generation. For outgoing replies, the latency will be small because the necessary keys were cached during the incoming request.

On the incoming data-path, the drive cannot begin decryption until it knows which key to use and the message authentication code cannot begin until at least a digest block worth of data is decrypted. When only protecting the integrity, a standard MAC algorithm will wait for the key lookup/generation latency. However, using the “Hash and MAC” approach described in Section 6.4, the drive doesn't need the key until the end of the MAC calculation so the lookup/calculation can be done in parallel with the hash value calculations, hiding the latency for most requests. For privacy, the drive must always wait for the proper key before decryption can start. These restrictions define when the cryptographic operations can begin.

Cryptographic primitives are the next component of latency. Encryption algorithms normally process 64 bit blocks, for 3DES, or 128 bit blocks for more modern ciphers. This small chunk (i.e., block) size allows encryption algorithms to form a fine-grained pipeline, producing results every 64 or 128 bits. An OceanLogic DES core processing one 64-bit block every 16 clock cycles [OceanLogic99], would implement 3-DES with a 48 cycle latency. This means one encryption block moves between *buffer1* and *buffer2* (Figure 7-2) in just 48 cycles. With no integrity, once a 64-bit block has been encrypted/decrypted, it continues onto the next pipeline stage for network transmission or storing to media.

If the drive is only protecting privacy, the latency of the cryptographic processing is the time to identify the proper keys and the time to encrypt/decrypt a single 64 bit block. For example, the OceanLogic DES core can process one input block, 64 bits, every 16 clock cycles [OceanLogic99], so 3DES could be implemented with about a 48 cycle latency. In functional pipeline shown in Figure 7-2, this 48 cycles is the time for the drive to move one encryption block between *buffer1* to *buffer2* on a write or *buffer2* to *buffer1* on a read. Since no integrity is being provided, once a 64 bit block has been encrypted/decrypted it is able to move on through the pipeline. Both *buffer1* and *buffer2* will only hold a single 64-bit block of data unless the encryption performance is limiting system throughput and queueing results.

When protecting integrity, MACing all the data substantially increases the latency of the cryptographic pipeline. Because MACs are only generated on discrete chunks of data, 64 KB in the prototype, even if MAC performance is not limiting system performance, the data will be buffered in the MAC computation until a result is generated. On outgoing traffic, a chunk of data can immediately be sent to the receiver but the receiver will be unable to process it until the corresponding MAC arrives which will not occur until the drive has finished MACing the entire chunk. Since the drive may be able to send data to clients faster than it can generate MAC's, the drive could stream data directly to clients and follow the data with the MACs. However, the drive could buffer data rather than sending it in order to preserve an ordering/interleaving relationship between data blocks and MACs as they are sent over the wire.

Adding integrity increases the latency of request processing more than encryption because the granularity of the functional pipeline stage is much larger. When privacy is used, the latency was simply the time to get the first bytes of output. When integrity is used, the message bytes may be available but they are not "acceptable" because there is no MAC value to either send or verify. In this case, latency corresponds to the time before the drive can generate a digest on a *digest-chunk*, the number of bytes between digests in the message stream, of bytes. In contrast to encryption where latency is the time to process the *first* encryption block in a message, the latency of a message authentication code is the time for it to process the *last* bytes in the digest-chunk.

Integrity processing latency varies by a factor of 20 or more depending on which MAC method is employed (i.e., HMAC-SHA1, HierMAC, HierMAC w/incremental digests), the size of the request, and the type of requests. Both HierMAC and HierMAC w/incremental digests (Figure 7-4), improve latency over HMAC-SHA1 by enabling data processing before the MAC key is identified. Both also use precomputed digests for some requests, reducing latency to a few iterations of the message digest calculation.

On writes, both HMAC-SHA1 and HierMAC have longer latencies than HierMAC w/incremental digests. HMAC-SHA1 latency is a function of chunk size while HierMAC depends on digest block size (Figure 7-4's example placed both sizes at 8 KB). HierMAC with incremental digests optimization reduces latency by enabling parallel computation over 256 B blocks, followed by the modular arithmetic (combining operators), and a final MAC. This parallelism does, however, require more hardware to process 256 B blocks in parallel.

For small requests, HMAC-SHA1's key generation dependency can create a long critical path. HierMAC avoids this critical path, allowing data computation to proceed without the key, but at the cost of an additional step, one extra iteration of the message digest calculation,

Finally, in addition to the cryptographic operations, the drive must also verify the nonce on a request check that the access credential is appropriate for the request. Checking the nonce requires searching for the nonce, probably in a hash table, to confirm that it has not already been received. The time for access credential checks depends on which of the choices described in Chapter 5 is implemented. For capabilities, the check requires only a few cycles to perform some simple comparisons. For metadata filters, the drive needs to execute the filter, which can take hundreds of cycles, and may need to perform additional expensive I/O operations. These checks can be performed in parallel with the cryptographic processing as long as the request is not irreversibly committed until all checks are completed and errors are properly prioritized so the drive does not leak information.

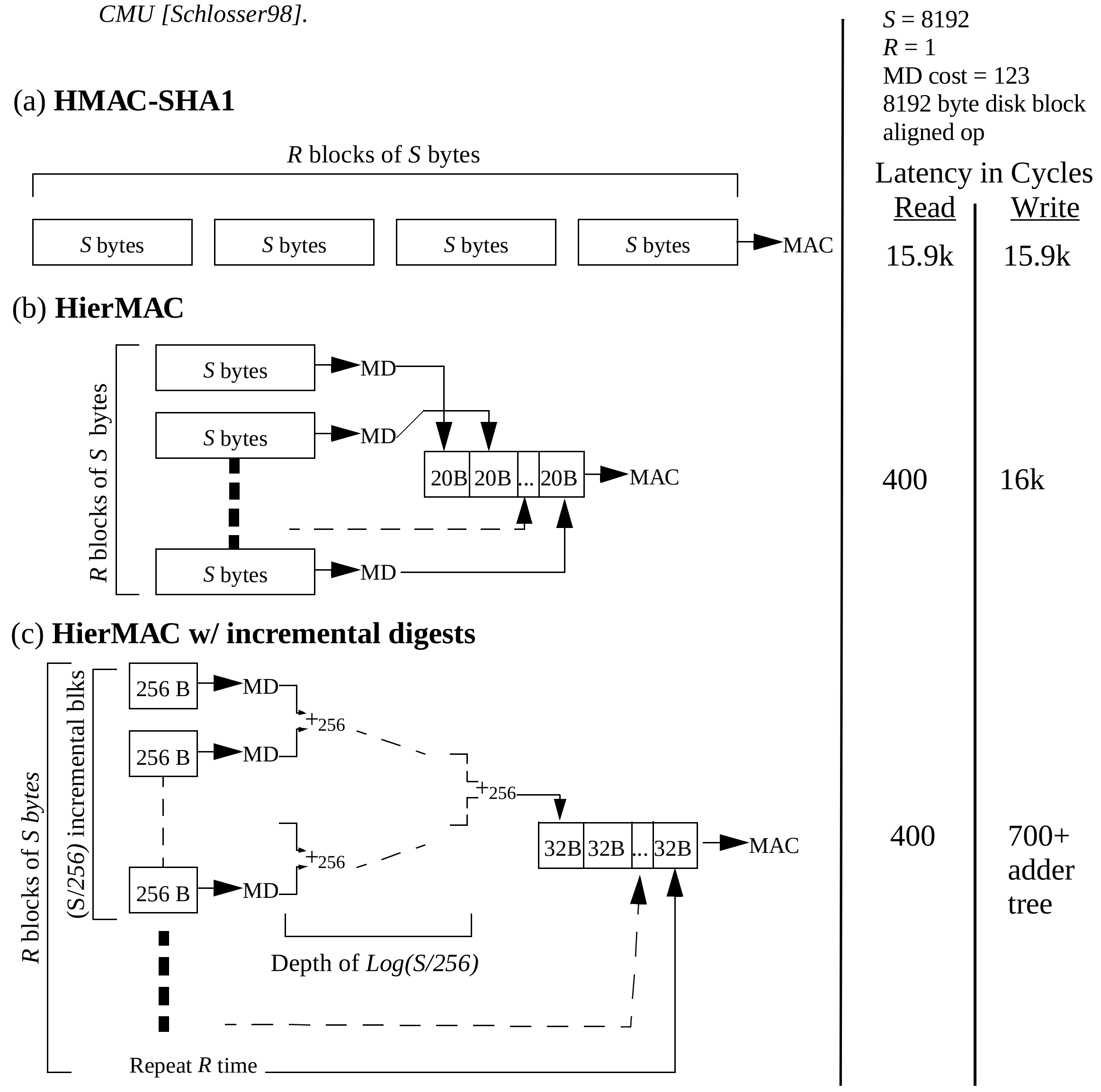
7.4.3 Throughput

Within a storage device, there are a variety of components that have different amounts of throughput. Where does security fit into this range? On one end of the drive, we have a high-performance full duplex network interface such as Gigabit Ethernet or Fibrechannel, which provides 1Gb/sec each direction. On the other end of the drive, the media transfer rates are currently at 28 MB/sec and they are increasing at 40% per year [Grochowski96]. Somewhere in between is the proper performance goal for cryptographic support. Clearly, if security doesn't even match the lesser of the two then the drive will fail to deliver some of its raw performance.

Figure 7-4 Comparison of Latency for Different MAC Approaches

*HierMAC uses precomputation and it has lower latency than HMAC-SHA1 on a read request. On a write request, incremental stored digests also reduce latency because it introduces more parallelism. This figure illustrates the critical path length, i.e. latency, for the three MAC approaches. All three approaches are parameterized by S, the size of the disk block, and R, the maximum number of disk blocks sent before a MAC is inserted. HMAC-SHA1 simply computes over R*S bytes, then produces a result. HierMAC can use precomputed digests on the read and it can compute the digests in parallel on a write (which is the same as HMAC-SHA1 when R=1). HierMAC with incremental digests has more parallelism which benefits small requests and writes, as well having the benefits of stored digests.*

On the right side of the figure, I list the latency to process a read and write of disk block, ignoring header and key costs. I assume S = 8192 bytes an R=1, which makes HierMAC and HMAC-SHA1 comparable on the write path. For per message digest latency, I estimate 123 cycles, which is the amount of time required per message digest block in an FPGA implementation of the SHA-1 core by Steve Schlosser and Ben Schmidt at CMU [Schlosser98].



If the cryptographic throughput matches the networking interface data rate, the drive will be able to read and write to its cache at the full network rates. From the security perspective, this minimizes the impact of an adversary swamping the cryptographic capacity of the drive with forged requests that are only recognized as forgeries *after* the MAC has been generated. Forging requests to the drive becomes a denial of service against network bandwidth rather than the drive's computational capacity since the drive has cryptographic resources are matched to the network.

A drive will only be able to accept a limited amount of writes, bounded by available on-disk RAM, at network rates because writes will need to be buffered to account for the much slower media data rates. With network rate cryptography, in the functional pipeline shown in Figure 7-2, these requests would be processed by the cryptography and wait in the work queue until they could be flushed to the media. With less than network rate cryptography, the requests would will queue in *Buffer1* until the cryptography could process them. However, once a drive verifies that a requests MAC is valid and decrypts the request, the drive can coalesce writes within on-drive buffers, which may reduce memory pressure if writes are small and sequential. For writes, as long as cryptography meets the media data rates, the impact of not having network data rate cryptography is which side of the cryptography a request will be queued. Before the cryptography, it is queued and unverified so the drive can't perform any media scheduling optimizations. After the cryptography, it has been verified so the drive can perform media scheduling optimizations but performance is still limited by media data rates.

Theoretically, a drive can service read requests at full network interface rates if the reads all hit in the data cache. Since drive memory is limited and small relative to media sizes, it is unlikely that a request will hit in the cache except for sequential accesses to the same object which benefit from disk block read-a-head and are still limited by the media rates. Normally, a sequence of reads will queue up as the drive waits to retrieve data blocks. If cryptography performance exactly matches media rates, the maximum throughput that a client will ever see from the drive will be media data rates. Cryptographic throughput should exceed the media rate to reduce the latency for servicing cache hits and to provide greater peak performance when cache hits do occur even though the sustained rate may be substantially lower. Additionally, exceeding the media rate allows requests to queue after cryptography, providing the drive with an opportunity to reorder requests and maximize its use of the media (although it can not exceed media data rates). The exact amount by which cryptographic data rates should exceed media rates will depend on the costs of increasing the data rates, the emphasis on peak bandwidth, and the probability of a cache hit. If a drive were to have a much larger data cache, optimizing for cache hits would be more compelling than the case for a drive with a few megabytes of cache. Other network attached storage devices like a RAID array may have large data caches and large amounts of aggregate media bandwidth so they would benefit from moving cryptographic performance nearer to network performance.

I have designed the security of NASD to maximize the parallelism available to the drive in order to improve its throughput. Encryption is highly parallelized because it uses counter-mode rather than more standard modes which have dependencies between encryption blocks. The message authentication code can also be parallelized at the granularity of disk blocks when using HierMAC and at the granularity of incremental blocks when using incremental hashing. The same features that make the computations parallelizable also enable the system to tolerate out-of-order reception while still performing the security processing in stream like manner.

There is nothing fundamentally preventing a drive from performing its cryptographic operations at full line rate. However, engineering a drive's cryptographic support to meet the peak data rates of the system implies that the drive is over-engineered for most of its workload.

7.5 Simulation Study of the Impact of Underprovisioned Digest Throughput on Client Latency when Protecting Integrity

I have argued that it is reasonable to have the cryptographic support run at less than network rates. In this section of the dissertation, through simulation, I explore the impact of reduced message digest throughput on the latency of filesystem operations as perceived by the end client. I will show that in filesystem workloads a drive can use a message digest functional unit that delivers as little as 1/3, 700 Mb/sec for Gigabit Ether, of its *duplex* network bandwidth while providing less than a 10% average increase in latency over no security.

In this analysis, I focus on the case of full integrity and no privacy being provided by the drive. Since integrity support in the drives is necessary for an application to run correctly on NASD while privacy can be provided at the application layer, I explore how little hardware support can be used to deliver good performance to clients. Furthermore, the finer pipeline stage of encryption/decryption implies it will have a smaller impact on latency than integrity processing.

Filesystem workloads are largely idle with occasional bursts. During periods of heavy load, client requests will be queued on some resources, which may be something other than the cryptographic support because the drive electronics can only handle a limited number of requests per seconds and, ultimately, the media is the final bottleneck. The digest throughput becomes a significant bottleneck but only when the drive electronics and media can support a large number of requests per second or in pathologically bad request mixes. Additionally, workloads rarely exploit the parallelism of a full duplex network interface which allows us to easily reduce our message digest capacity to the simplex rather than duplex bandwidth. In the remainder of this section, I will describe my simulation study and illustrate this argument holds true.

7.5.1 Simulation Environment

For this simulation, I used a trace from a University of California, Berkeley Auspex NFS fileserver [Dahlin94] and a trace of the Carnegie Mellon University Parallel Data Lab's AFS server collected in early 1999. Both of these traces are described in Section 5.1.3.1.

Each filesystem level request is mapped to one or more NASD level requests as shown in Table 7-1. For AFS BulkStatus, AFS RemoveFile, and NFS DeleteWrite, these requests were issued to the drive at line rates because there are no data dependencies between the operations. However, for an AFS StoreData, AFS CreateDir, AFS MakeDir, and NFS DirRW, the later NASD operations could not be issued until the first NASD operation has completed because of dependencies on either the data being written or a new NASD ID. I modeled operations that would read or write a stored directory object as 8K read or write operations because 8K is large enough to hold most directories and allows the system to exploit the stored digest optimization presented in Section 6.4. For some of AFS FetchData and StoreData operations, no size information was recorded in the trace so I use the average FetchData and StoreData sizes reported by the server through the x-stats interface during the tracing time period.

Table 7-1 AFS and NFS to NASD Request Mappings

Each filesystem level operation is converted into one or more NASD operations. The following table describes this mapping:.

AFS Operation	NFS Operation	NASD Operation(s)
FetchData	Read Block, DirRead	Read
StoreData	N/A	Write + SetAttr
N/A	Write Block	Write
FetchStatus	Read Attr	GetAttr
BulkStatus	N/A	*N* * GetAttr
N/A	DirRead	8K Read
StoreStatus	Write Attr	SetAttr
CreateFile, MakeDir	DirRW	Create + 8KWrite + SetAttr
Rename	N/A	8K Write
FetchACL	N/A	8K Read
Link/SymLink	N/A	8K Write
RemoveFile	DeleteWrite	Remove + 8K Write

Both the AFS and NFS workloads are an approximation of the workloads I expect to be offered to a NASD drive. A real NASD drive may not handle the equivalent of an entire fileserver's namespace stored on a single device. However, if an AFS or NFS system is run on top of NASDs, I expect the distributions of requests to remain roughly the same so the workloads are a good first approximation of the workload that would be presented to a drive.

Each NASD operation has a fixed NASD header as well as arguments and a result structure, which are shown in Table 7-2, in addition to any data being read or written. The simulator accurately models their cost as well as the networking cost of a 32 byte header (approximating a small UDP and RPC header per request).

The simulator models the message digest cost of using HMAC-SHA1 and applying integrity protection to both the arguments and the data using the simple capability model of access credentials, so access credentials are small fixed sized fields. Pessimistically, I assume that there is no cache of capability keys and they must be generated on each request.

The simulator uses queuing models of three classes of drive resources: network (input and output links), message digest unit (SHA-1), and the drive electronics as shown in Figure 7-5. Each network interface has a gigabit of available bandwidth. The SHA-1 unit is characterized as having *n* bits/second throughput. Unless otherwise noted, a single message digest unit is modeled on a drive which handles all the digesting required for requests, replies, and key generation.

Table 7-2 NASD Argument and Result Sizes

This table describes the sizes of the arguments and results for the major NASD operations. In addition to the values in the table, every request argument includes an 8 byte security header (Section 4.4.1), a 48 byte public access credential, implementing a capability model, a 20 byte MAC, and an 8 byte timestamp. Each result will also include a 20 byte MAC and an 8 byte timestamp. NASD attributes are 336 bytes long.

NASD Operation	Argument Size in bytes	Result Size in bytes
Create	64	4
Remove	16	4
SetAttr	352	344
GetAttr	16	344
Write	104	8
Read	104	8

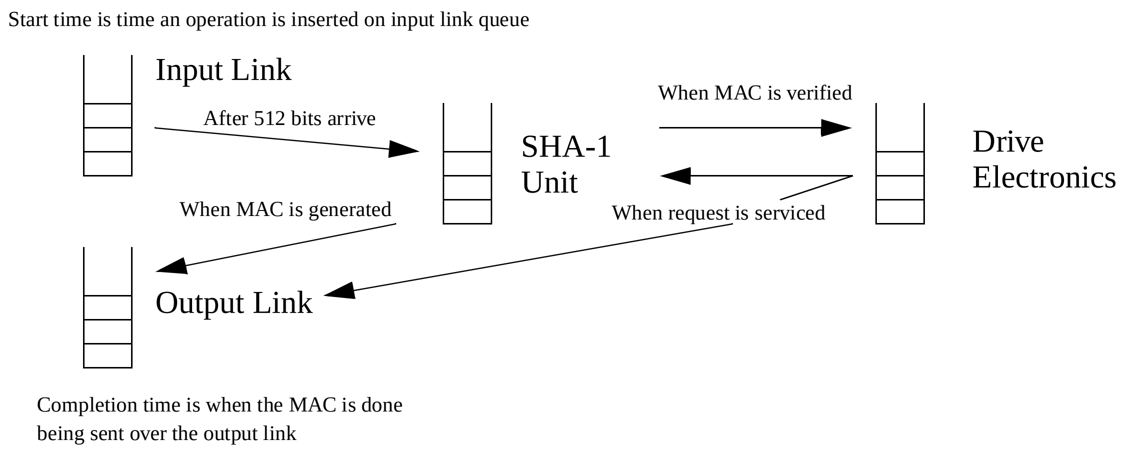

Start time is time an operation is inserted on input link queue
Input Link
After 512 bits arrive
When MAC is verified
SHA-1 Unit
Drive Electronics
When MAC is generated
When request is serviced
Output Link
Completion time is when the MAC is done being sent over the output link

Figure 7-5 Simulation Queueing Model

These are the four resources modeled in the simulation and the transitions of requests between the queues.When a client sends a request to the drive, the request is first placed on the drive's input link queue. After 512 bits of data have been transferred over the link, the drive has enough data to begin SHA-1 and the request is placed on the SHA-1 queue. When all the required SHA-1 work is complete, the request's MAC has been verified and the request is queued on the drive electronics. After the request is serviced, the result is queued on both the SHA-1 unit and output link to concurrent send and generate the reply MAC. If the data is completely sent to the client before the reply MAC is generated, the MAC will be enqueued separately on the output link when it is complete.

I use a simple model of the raw drive functionality in which all requests have a fixed service time and media time is ignored. The base service time is 0.12 milliseconds which is the time for a relatively modern Seagate Ultra Wide ST34371W drive takes to process a prefetch hit minus the time spent on the SCSI bus, i.e. the request "think time" [Riedel98b]. This limits the drive to a maximum of 8333 requests per second. By eliminating seek time and internal data transfer times, I bias heavily against reducing message digest capacity because the delay due to slower cryptography is more significant when the slowest portion of the drive is ignored. In some sense, I am modeling a solid-state disk while, for the foreseeable future, most NASDs will use magnetic media as the backing store.

From the perspective of clients, the primary impact of reducing the message digest bandwidth, assuming the SHA-1 unit can sustain media rates, will be seen as added latency, compared to a system with no security, on each request. In this study, I compare various points in the design space based on the added *percentage* latency per request. I selected this metric because it accounts for the fact that a delay of 0.03 milliseconds will have a substantial impact on a GetAttr, or other small requests, but will have a much less noticeable impact on a long-running request such as a 64 KB write.

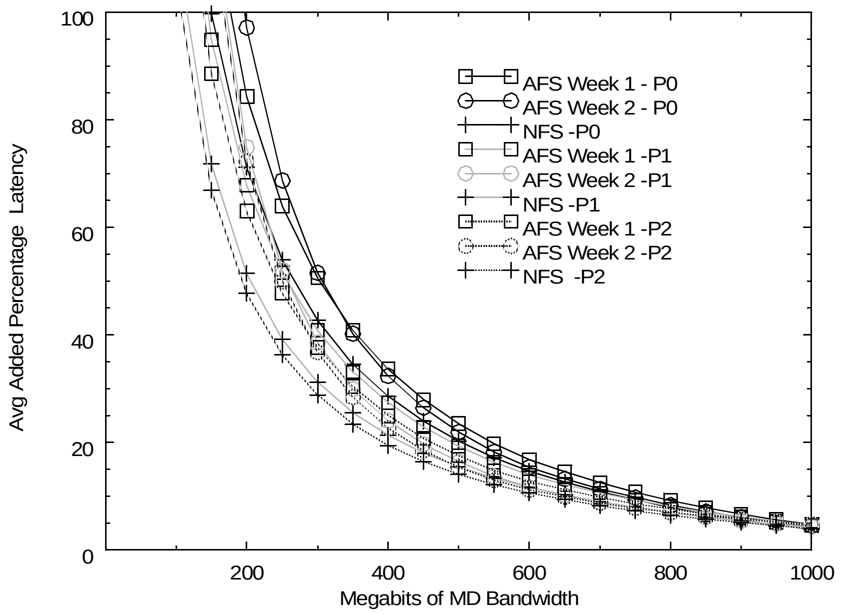


Figure 7-6 Average Additional Latency Seen by Clients

For all workloads, a drive with only 700 Mb/sec of message digest bandwidth adds an average of less than 10% additional latency to filesystem requests compared to their latency without security. These simulation results show the impact of having less message digest bandwidth than the full duplex network bandwidth(2Gb/s) for the three sample workloads. The x-axis shows the throughput of the SHA-1 unit and the y-axis marks the average percentage increase in latency of a filesystem request in comparison to the same request running without security. The additional impact of applying the precompute optimizations is also shown. P0 is using no precompute, i.e. all bytes are MAC'd using HMAC-SHA1. P1 is using stored message for each 8K disk block to reduce the computation on a read operation. P2 adds a stored message digest for attributes to reduce computation on GetAttr operations.

7.5.2 Results

Figure 7-6 shows that a drive with only 200 Mb/sec of message digest bandwidth adds an average of more than 100% latency to filesystem requests compared to their latency without security. In contrast, a drive with 700 Mb/sec of message digest bandwidth adds an average of less than 10% additional latency. Figure 7-6 shows the impact of underprovisioned SHA-1 capacity and the impact of precomputing hashing over nothing, disk blocks, and disk block + attributes. Without any optimizations (P0), reducing SHA-1 bandwidth to 600 Mb/sec adds an average of 20% more latency to each request. Precomputing stored digests and using HierMAC (P1) reduces the added latency by about 5% for the 500-600 Mb/sec range while precomputation for attributes (P2) reduces it by another 1%. These improvements hold for all three of the workloads. Overall, the cryptography had a much larger impact on the AFS workloads compared to the NFS workloads because the AFS workloads include 64 KB writes while NFS performs only 8 KB or smaller transfers. Reducing SHA-1 bandwidth to 550 Mb/sec and using all the

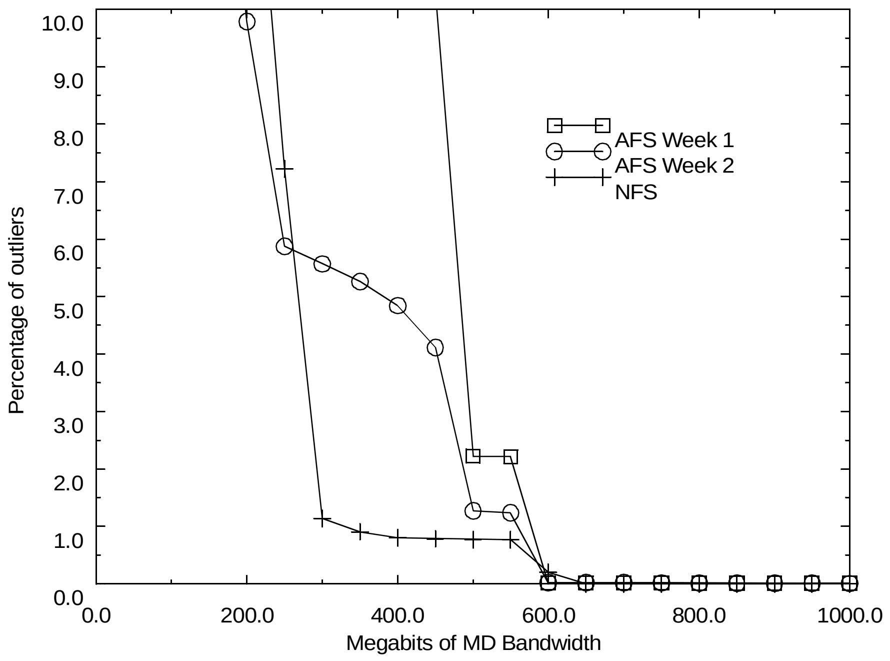


Figure 7-7 Percentage Outliers

If message digest bandwidth is less than 500 Mb/s, a large number of requests take twice as long, i.e. outliers. However, this quickly converges to almost no requests being outliers. The x-axis is the throughput of the SHA-1 unit and the y-axis the percentage of filesystem level requests where the request service time was twice as long as the time with no-security i.e. requests where the added latency was at least 100%. These are the periods when clients are most likely to noticed the added latency. These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

precomputation optimizations results in an average of less than 15% latency being added to every request. Since I am exploring how far I can reduce the requirement for SHA-1 bandwidth, all subsequent simulations in this chapter were run with precomputation used on both data and attributes.

Reducing the SHA-1 throughput introduces a bottleneck with queuing into the system which translates into variability in service times seen by clients. I measure this variability as the percentage of requests, which I call *outliers*, that take more than twice as long as the same request would have taken with no security. These are the requests for which clients are most likely to notice that service time has increased. I did not to use variance because variance can be highly skewed by a few extreme points and, when constrained by SHA-1 bandwidth, there will always be times when queuing on the SHA-1 unit substantially impacts overall request service time. For example, when small requests are queued behind a write request (i.e. the head of line problem), the small request will suffer a much larger percentage change in service time than the write suffers even though

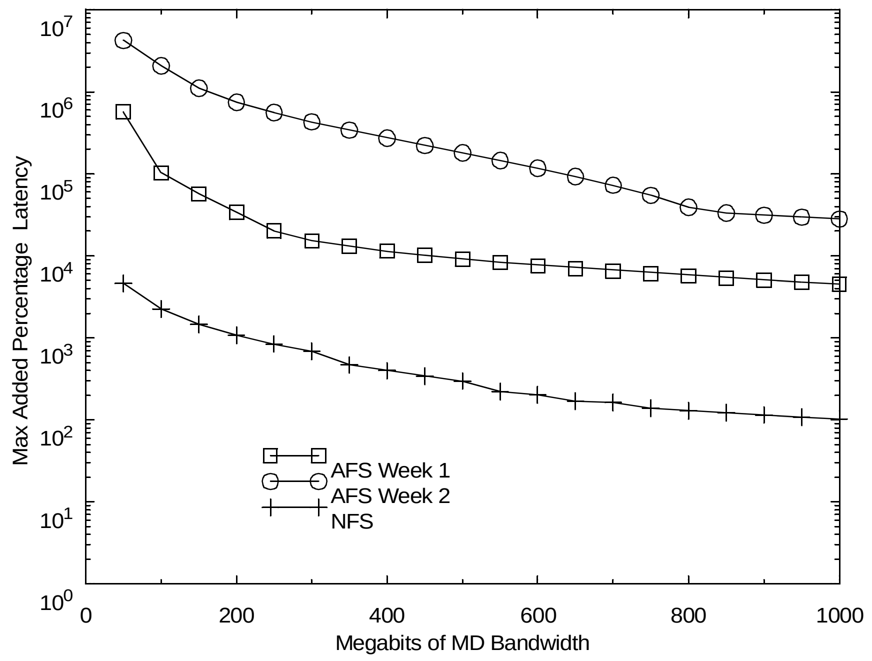


Figure 7-8 Maximum Additional Latency Seen by Clients

Since SHA-1 bandwidth introduces another potential bottleneck, there will always be cases where it introduces queuing and some request takes much longer than normal. The x-axis shows the throughput of the SHA-1 unit and the y-axis marks the worst case added percentage latencies for each trace in comparison to the non-security version of each request. The worst-case is significantly better in NFS because transfers are smaller. The AFS maximums illustrate that the worst scale can vary substantially from trace to trace even in a single filesystem and single user environment.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

most of the delay is spent servicing the write. For this analysis, it is useful to talk about how often the client is likely to notice that performance has substantially degraded which the counting the percentage of outliers captures.

When SHA-1 throughput is badly mismatched for its workload, a large percentage of the requests take more than twice as long but this quickly converges to less than 1% as capacity is increased to meet demand (Figure 7-7). For 500 Mb/sec or less, 2.5% or more of the requests are outliers but, for 600 Mb/sec or more, less than 0.03% of the requests are outliers.

For each workload, there are times when performance degrades substantially as a result of the SHA-1 bandwidth becoming the bottleneck (Figure 7-8). While the average added latency and, to a lesser degree, the percentage of outliers is relatively similar over the two AFS traces, the worst cases can differ by a factor of 10. The outliers and the worst cases are partially caused when a large request prevents small requests from making progress through the functional pipeline. Since AFS uses larger data transfers, this effects has a larger impact in AFS workloads than NFS workloads. Even as capacity is increased, the worst case does not change as quickly as the average within the limits I explored because there are always some brief periods where the entire system wants to run near saturation and substantial queueing can quickly occur. I explore two ways that a drive could reduce this effect: time-slicing the message digest unit or pulling data.

In the initial simulations, requests queued up for SHA-1 work are processed in order and each request is processed to completion. An alternative is to time slice the SHA-1 resources and prioritize small requests. The next series of simulation results explores this approach. The SHA-1 resource has two queues: a high priority queue for all operations requiring SHA-1 processing on *N* bytes or less and a low priority queue for the rest of the requests. An SHA-1 operation is the amount of SHA-1 work necessary for a request on the incoming or outgoing path. So, the processing of a read request and the processing of all the data in the result are separate SHA-1 operations. When the SHA-1 resource is free, high priority operations are processed first. If no high priority operations are available, a low priority operation is processed and allowed make *N* bytes of forward progress before it is put back on the head of the low priority queue and both queues are re-examined.

Allowing this simple prioritization and preemption significantly curtails the maximum wait a request may have due to being backed up behind a larger request (Figure 7-9) and also reduces the outliers (Figure 7-10). For AFS workloads, with their larger operations, the time-slicing approach significantly improves the worst case since a small request will spend less time stalled behind a large request. However, if the time-slicing interval is too small, large requests become starved during periods of heavy activity and the worst case degrades. Time-slicing of the SHA-1 resource does not improve the number of outliers significantly. In the slow cases, some write operations are now being starved into becoming outliers. In the fast cases, there are already very few outliers so the improvement is not very significant.

Another solution is to place the drive in control of its data movement. The networking communities' solution to preventing a sender from swamping a drive with data is to allow the drive to control the write request and *pull* the data from the client rather than allowing the client to *push* the data at its convenience. Logically, pull semantics place the most resource poor of the two parties in control of bandwidth allocation decisions. In addition to allowing the drive to schedule data arrivals to meet its buffering needs and more carefully match media rates, the drive can also schedule based on the availability of the SHA-1 resource.

To approximate pull semantics, I simulate the drive synchronously handling only the control portion of the write by mapping all writes to 0-byte writes. Figure 7-11 shows that this approach significantly reduces worst case behavior as well as the number of outliers thus should improve the variability perceived by clients. For all workloads, applying pull semantics to the drive reduced the number of outliers by at least a factor of 4 for all systems with 400 Mb/s of SHA-1 bandwidth or more and reduced it to zero for 600 Mb/s or faster systems. This reinforces the idea that writes are a major issue for a drive. For the both AFS workloads, the percentage of outliers was constant from 400 Mb/s to 550 Mb/s,

Figure 7-9 Maximum Latency with Time Slicing SHA-1 Unit

The simple time slicing approach significantly improves the worst case for AFS workloads. For each of the workloads, I compared the non-slicing case against preempting every 16K, 8K, and 2K bytes to allow smaller jobs to be processed and using the same cutoffs to distinguish between low and high priority operations.The x-axis shows the throughput of the SHA-1 unit and the y-axis marks the worst case added percentage latencies for each trace in comparison to the non-security version of each request.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

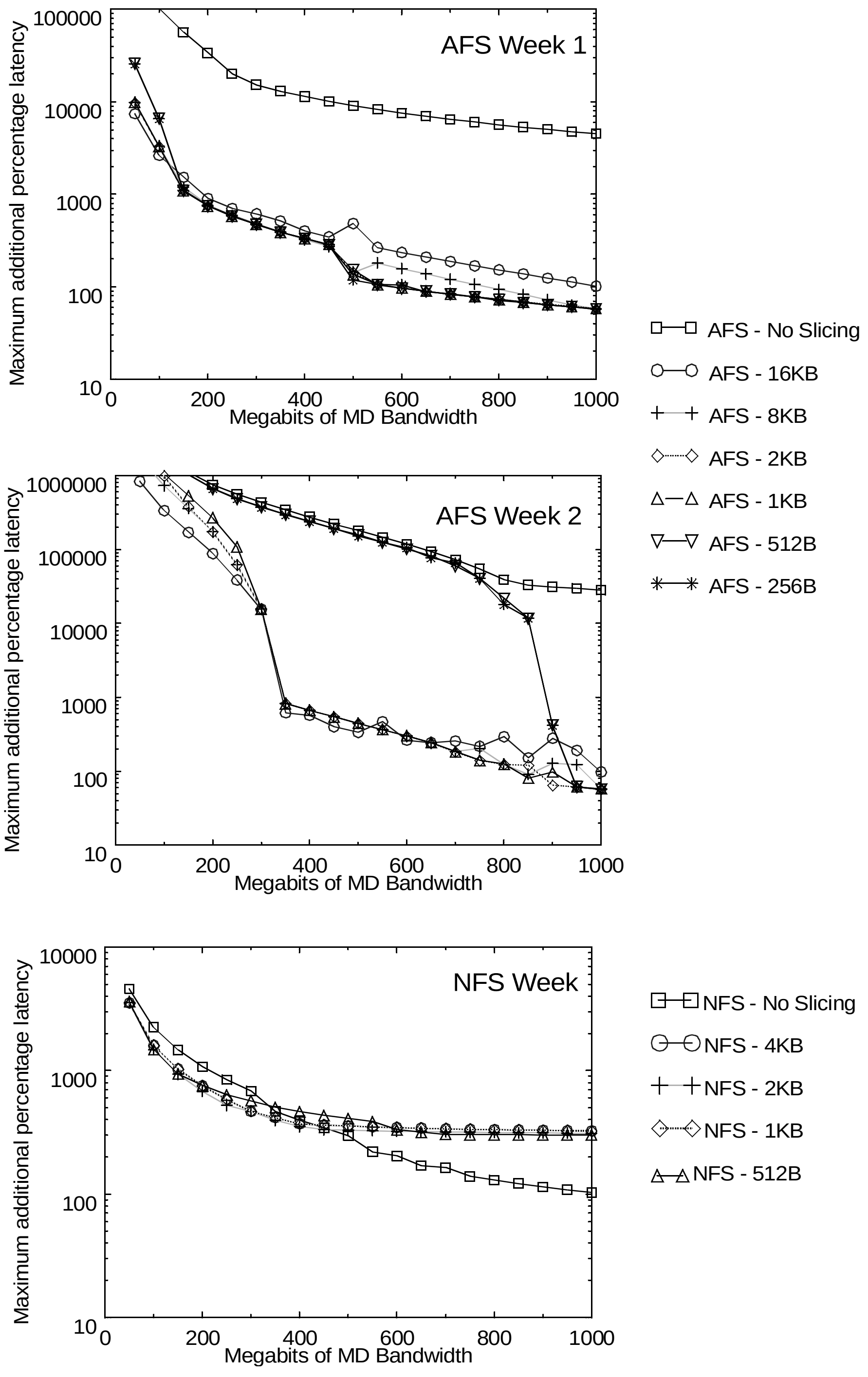


Figure 7-10 Impact of Time Slicing SHA-1 Unit on Percentage of Outliers

Time slicing has a negligible impact on the percentage of outliers. Note, in contrast to earlier graphs, the Y-axis is in log scale. The x-axis shows the throughput of the SHA-1 unit and the y-axis marks the percentage of filesystem level requests where the request service time was twice as long as the time with no-security. For each of the workloads, I compared the non-slicing case against preempting at fixed intervals to allow smaller jobs to be processed and used the same cutoffs to distinguish between low and high priority operations. For each of the workloads, I compared the non-slicing case against time slicing at fixed byte intervals to allow smaller jobs to be processed. These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

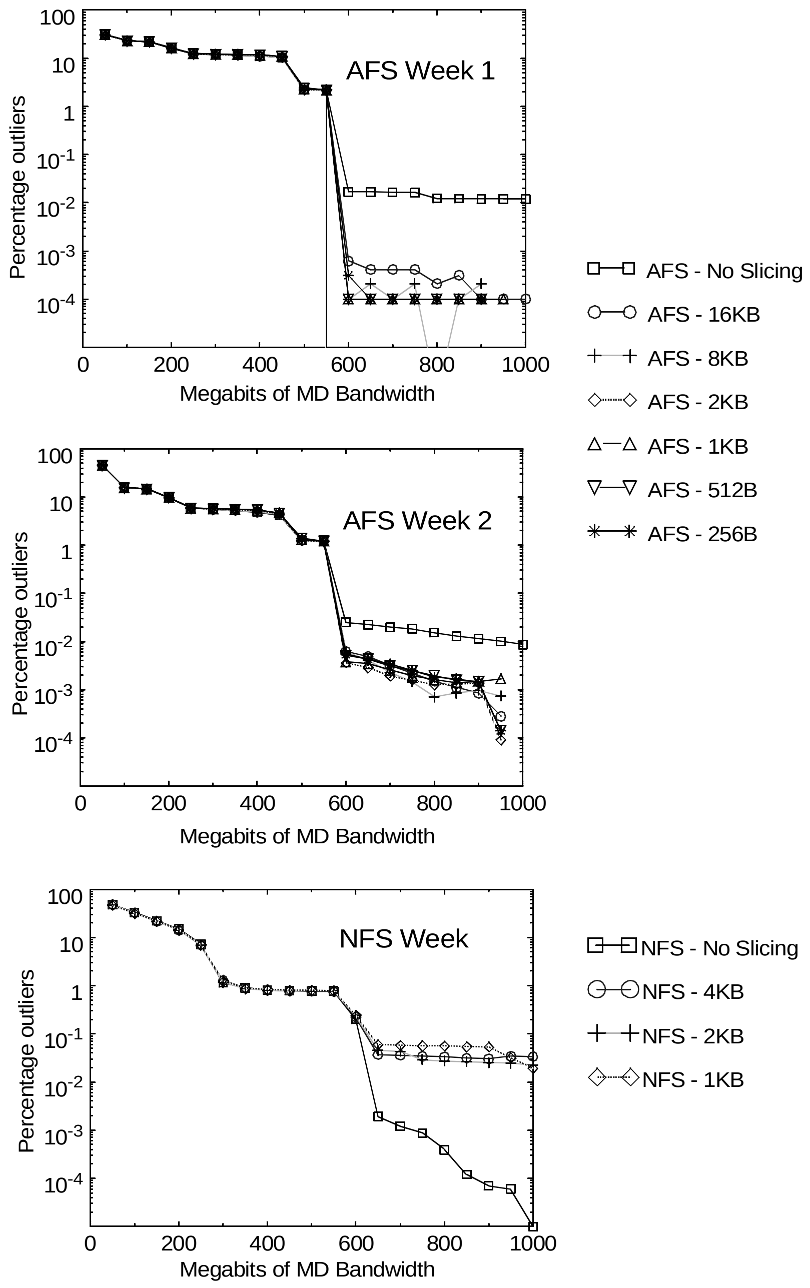


Figure 7-11 Impact of *Pull* Semantics on Added Latency

Pull semantics reduce the percentage of outliers and improve the worst case for all three workloads. All write operations are mapped to 0-byte writes to approximate the drive synchronously handling the control portion of a write but being able to schedule the data processing to minimize the impact on other requests and efficiently utilize its buffers and limited media bandwidth. On the left, the x-axis shows the throughput of the SHA-1 unit and the y-axis, percentage of filesystem level requests where the request service time was twice as long as the time with no-security, that is, requests where the added latency was at least 100%. On the right, the x-axis shows the throughput of the SHA-1 unit and the y-axis marks the worst-case added percentage latencies for each trace in comparison to the non-security version of each request.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.

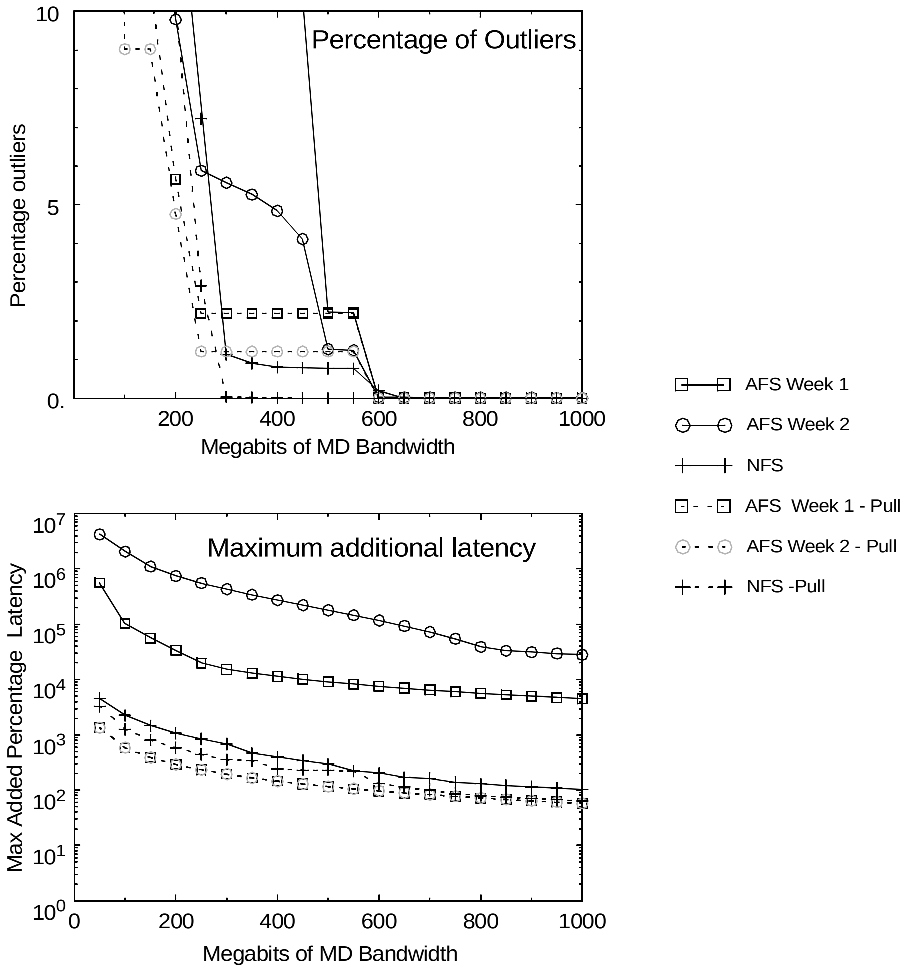
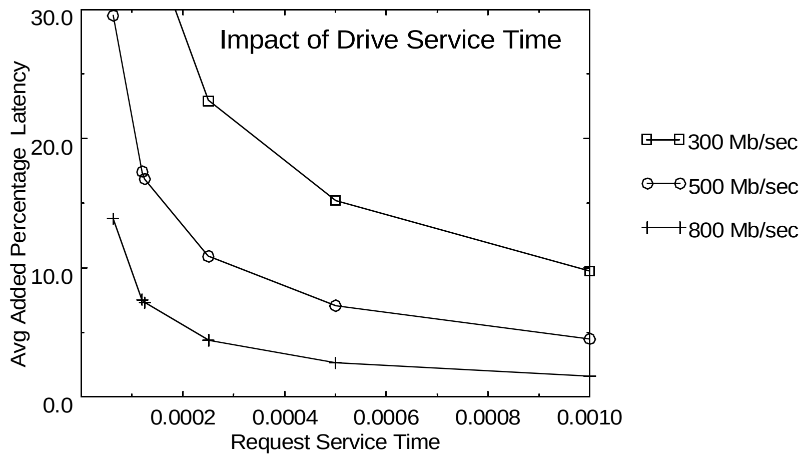


Figure 7-12 Sensitivity to Drive Service Time

A small request service time, the time spent in the drive electronics in the queuing model shown in Figure 7-5, increases the impact of decreased SHA-1 throughput. This figure shows the first week of the AFS workload at three levels of SHA-1 performance. The x-axis shows the request service time in seconds and the y-axis marks the average additional latency of a filesystem request in comparison to the no security case.These simulations are for the full integrity case using HierMAC and precomputed SHA-1 digests on disk blocks and attributes.



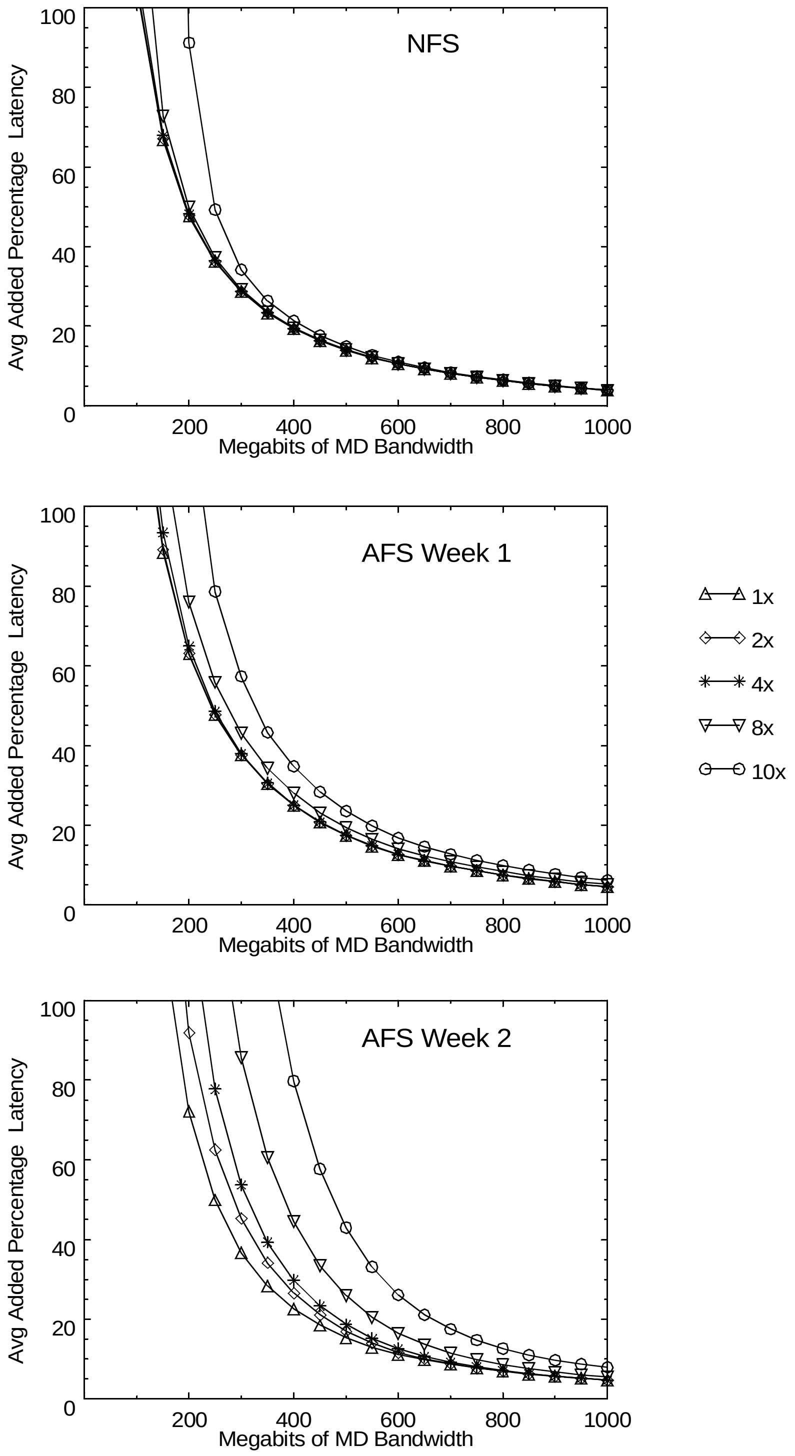
which indicates there are still some cases where substantial queuing can occur because of a large number of quickly issued requests — most likely, bulkstatus requests. Figure 7-11 shows that reducing writes to control operations has had a dramatic impact on the worst case scenario for the AFS workloads, with their larger requests, compared to the effect on NFS.

7.5.3 Sensitivity Analysis

The simulation is sensitive to the drive's basic response time, the time spent in the drive electronics, particularly for the smaller requests. If a drive can only handle 1000 requests per second, the amount of time spent in a slow message digest unit or on the wire is small relative the total service time of the request. However, if a drive can handle 32,000 requests per second, the digest time and wire time are a more substantial portion of the service time. For all the previous simulations, I assumed a drive could service 8333 requests per second. In Figure 7-12, I show how the service time of the drive affects the average added latency seen by clients. This simple model illustrates that queuing on the drive internals (which, in a real drive, includes media time) has a significant impact on how little SHA-1 throughput is acceptable. For a drive servicing only 2000 requests per second, the average additional latency with 500 Mb/s of SHA-1 bandwidth is less than 10%. Since my simulations assumed an aggressive number of requests per second, this less aggressive value indicates that the drive actually services fewer requests per second

Figure 7-13 Effect of Increased Load on Added Latency

These graphs show how decreasing the inter-arrival time between requests by a constant factor impacts the added latency. The added latency is measured with respect to the service time the same request, with the reduced inter-arrival time, would incur. For all workloads, somewhere between 8x and 10x the non-security case starts incurring large queuing delays.



because it is touching the media and will have a smaller latency penalty for reducing SHA-1 digest.

Increasing the load on the drive exacerbates the impact of reducing SHA-1 bandwidth (Figure 7-13). I reduced the inter-arrival delay between subsequent requests in the traces by a factor of two, four, eight, and ten to understand how increased load affected the system. The NFS trace shows an interesting effect, as the load increases beyond a certain point, the impact of underprovisioned SHA-1 bandwidth starts to decrease. In the figures, I am comparing a request's service time with security versus without security at the same level of load. During periods of sustained heavy load, requests are being queued for long periods of time in the non-security case so a small additional penalty due to queuing for the SHA-1 unit is a decreasingly small portion of the overall service time. For slower drive speeds (6250 requests per second), the same effect shows up in AFS workloads but doesn't appear in the baseline drive (8333 requests per second). The effect appears to be highly dependent on a balance between the request distribution, inter-arrival time, and request service time. When this queueing effect does not occur, the additional load decreases the dead time between requests which gives the SHA-1 units less time to use to compensate for differences between SHA-1 and network bandwidth. As a result, the slower SHA-1 cases suffer more from the increased load than the faster cases.

7.6 Hardware Solutions

Hardware support for cryptography can take different forms: an additional cryptographic ASIC, expanding the role of an existing ASIC, or adding an FPGA. The cryptographic ASICs that I discussed in Section 7.3 provide a baseline estimate of performance and an upper bound on the cost. Adding a another ASIC to a drive adds more cost than necessary as well as another point of failure on the drive. A more likely path for drive manufacturers is to expand the role of the central drive ASIC, which I will discuss in the Section 7.6.1. However, both of these solutions suffer from fixing the cryptographic algorithms in silicon and they can not adapt and deliver high performance if a cryptographic algorithm needs to be changed. While software does not suffer from this mutability problem, software only solutions cannot deliver the necessary performance without a powerful CPU. In Section 7.6.2, I will discuss reconfigurable hardware technology, though while still in its infancy, offers an appealing technology that can deliver most of the performance of hardware coupled with much of the flexibility of software.

7.6.1 Expanding the Role of the Integrated ASIC

With the spreading use of cryptography, many vendors have produced cryptographic logic cores for common algorithms which could be used to add cryptographic functions to a drive ASIC. SICAN Microelectronics Corporation sells an SHA-1 logic core requiring approximate 20,000 gates [SICAN99] and Asic International sells an SHA-1 logic core which can deliver 200 Mb/s throughput with a 80 MHz clock [Asic99]. Similarly, Xentec licenses the Ocean Logic Pty Ltd.’s DES core which can deliver 400 Mb/s with a 100 MHz clock using 3,500 gates in a LSI 500K technology [OceanLogic99] for a $30K unlimited license fee [Wania99]. Triple-DES versions are also available. It is certainly feasible to purchase off-the-shelf logic cores for the computationally expensive cryptographic operations and integrate them into the drive using under 35,000 gates.

Integrating the cryptography onto the central ASIC is cost-effective because it does not require additional chips, although it may reduce the yield of the primary ASIC. More importantly, cryptographic functions can access the data as it passes through the central ASIC while the data is stored in on-chip SRAM[1] rather than contending for the slower DRAM which is already a bottleneck in the drive. However, if the cryptography runs significantly slower than the network interface, the drive will need to buffer requests in DRAM, thus increasing memory bandwidth pressure when the drive is under a heavy load.

On read requests, the data passes through the primary ASIC on its way out to the network so the primary ASIC could do the cryptography as the data moves from local SRAM to the network interface. On the write requests, data passes through the primary ASIC as it goes from the queue in DRAM out to the R/W channel. So, the ASIC is also an appropriate location for cryptography.

When the cryptographic functions are implemented in the ASIC, they can never be updated. In the unlikely event that the encryption or message authentication code is broken, the only option is to use software cryptography through a firmware update, or use the broken algorithms. If the drive was designed with a fast processor and software cryptography rather than implementing the cryptography in silicon, the processor performance will transfer to any new algorithm that is used. Between the flexibility of software and speed of an ASIC is reconfigurable hardware, which is an alternative solution.

7.6.2 Potential of Reconfigurable Hardware

Field Programmable Gate Array technology (FPGA) is a young technology that provides some of the performance of a custom ASIC and some of the flexibility of software. An FPGA is a programmable interconnected mesh of logic blocks which can each be programmed to perform one of a set of simple functions. Together, the hundreds or thousands of logical blocks on the FPGA can be connected to implement complex functions.

In recent years, reconfigurable hardware has been used to implement many cryptographic algorithms such as DES [Kean98, Luk97], REDOC III [Guerro95], IDEA [Budiu99], and MD5 [Arnold98]. The same logic cores used to implement

1. In addition to being closer to the logic, SRAM is generally faster and more available than DRAM. Thus, it is advantageous to process the data while it is in the on-chip SRAM [IBM97].

cryptography on an ASIC can be used in an FPGA. For example, the Ocean Logic DES core can deliver 308 Mb/sec on a Xilinx Virtex running at 77 MHz [OceanLogic99] and IBM has implemented SHA-1 in an FPGA running at 200 Mb/sec. While FPGAs lag behind an ASIC in performance, they deliver much greater performance than software and much of software's flexibility.

Key-specific optimizations further reduce the performance difference between FPGA and ASIC technologies by configuring an FPGA to compute with exactly one key [Budiu99, Luk97]. This can deliver a 30% speedup and reduce the size of the circuit by half [Luk97]. While key-specific optimizations are appealing, current FPGAs can not be reconfigured quickly enough to change keys on a per-request basis, which is necessary for NASD. Current research into incremental FPGAs offers an alternative that reduces reconfiguration penalties by only changing the portions of the logic that are key specific [Schmit97] and reducing FPGA compilation times [Budiu99]. If rapid reconfiguration and quick compilation become a reality, reconfigurable computing will be a promising way of delivering both performance and flexibility for security components.

Programmable hardware introduces the risk of an adversary modifying the cryptographic algorithms. If an adversary can transform the encryption into a null operation or have the message authentication code generate a predictable value, then the adversary has broken the security. When the algorithm is implemented in an ASIC, an adversary can not easily change the algorithm. However, normal access control mechanisms can prevent an adversary from convincing the drive to reprogram the FPGA if the implementation is done carefully.

FPGAs are not yet high-volume commodity devices. Any manufacturer who used a high-performance FPGA for a performance-critical function on a drive will pay a cost premium for the flexibility. However, the FPGA market has been growing steadily over recent years and as demand grows and the technology matures, the cost will fall and FPGA solutions may become more appealing in the future.

7.7 Chapter Summary

Security processing introduces an additional functional step in a drive's internal pipeline. Within security processing, the decision to protect integrity, privacy, or both affects how security creates latency. Encryption operates on a fine-grained pipeline and it has a small impact on latency. Message authentication codes introduce results at discrete intervals, which increases latency. The different message authentication code approaches have different impacts on latency. HMAC-SHA1 has a long critical path, while HierMAC requires less work and less latency on large disk-block aligned reads. HierMAC with incremental digests can operate on 256 byte blocks in parallel which reduces the critical path and corresponding latency.

A network attached storage device has two core data rates: the network interface data rate and the media data rate. In the best case, the storage device can service reads out of cache and absorb writes, up to the limitations of its buffers, at full network data rates. Eventually, write operations will be flushed to the media and some reads will not hit in the cache. These data rates define the bounds of a range of security performance that are interesting.

Clearly, faster is better but faster is also more expensive. I’ve shown that some cryptographic support is not unreasonable in cost although the exact cost and performance is dependent on the levels of integration and fabrication technologies. In simulation, I’ve shown that, when providing full integrity to requests, filesystem workloads require message digest support which achieves only 1/3 of the drive’s full duplex network performance to service all requests with an average of less than 10% additional latency.

Chapter 8: Tamper Resistance

So far, I have emphasized a drive's ability to both provide data, as part of a secure application, and protect network communication. Using the basic design presented in Chapter 4, a filemanager can enforce its policies over a drive while both the drive and client cooperate to protect the integrity and/or privacy of communication. In this chapter, I discuss how to defeat attacks by an adversary with physical access to the drive.

One approach is to use a secure facility and assume that an adversary will not gain physical access to server machines or drives. Tamper-resistance technology offers an alternative that allows devices to be secure without being behind locked doors. Past research has demonstrated that tamper-resistant technologies can be used to provide a trusted and secure processing environment within a normal workstation or server [Weingart87, White87, Yee93, Yee95, Smith98]. Similar technology could be applied to storage devices. However, tamper-resistance technology has not been examined in the context of storage devices which require both high security and low cost as well as having their own particular internal architectures.

If a storage device is tamper resistant, it can keep long term secrets. This allows it to safely manage multiple keys, including keys used exclusively to encrypt data on the media, while preventing an adversary from extracting the keys. Without tamper resistance, an adversary with physical access to a device could extract its cryptographic keys and thus the storage device encrypting data on the media would not add any security.

First, I describe the standard machine room approach to physical security. Next, I present background information on tamper resistance and modern attack techniques. I also show, at a high level, how the tamper-resistance technologies must be applied to a disk drive to make it secure. Finally, I describe the advantages of having a storage device encrypt data as it is written to the media.

8.1 Machine Rooms

The simplest way to prevent an adversary from gaining physical access to storage devices is to keep them in a secure facility. In a secure server attached disk system, the server machines are normally stored in a locked room which is continuously monitored by either security personnel or system administrators. If an adversary attempts to tamper with or steal a disk, she will hopefully be detected. If the machine room is small, the cost of continuous monitoring and maintaining physical security will add a significant premium to the storage system. But, if the machine room is large, it becomes easier for someone, perhaps an employee authorized to be in the machine room, to slip in and steal or modify a small storage device, e.g. a rack mounted disk drive, and escape before the loss is detected. However, careful planning and a well defined access control policy for a machine room can minimize this risk.

The small size of the drives makes them particularly susceptible to theft. Unlike a workstation which, due to its size, is difficult to steal from a machine room, a drive that is plugged into a network port can be easily hidden in a briefcase or backpack. A drive’s portability implies the necessity for more careful monitoring than a site needs for server machines in order to obtain comparable levels of security.

8.2 Tamper Resistance

The protection of a storage device’s cryptographic keys and all key-dependent calculations is necessary to make strong assertions about the security of a NASD storage system. If an adversary can physically access a storage device, she can probe the device and extract the keys necessary to impersonate it as well as read all data stored on the media. If the drive’s location has acceptable levels of physical security then the storage system only needs secure protocols. But, if the location is not secure, such as in an office or widely accessible machine room, then the device must provide some of the physical protection.

For some environments, a physically secure device is more appealing because it reduces the security assumptions to characteristics of the device and its management rather than requiring a secure operating environment. By making security an attribute of the storage device rather than of the environment, the drive can have high levels of security despite being in an insecure environment. For example, a user may purchase a new storage device and install it directly on the network in their office but still have it centrally managed by a remote filemanager as part of a secure application. Because a user has a sense of ownership for the device, she wants to have physical control of it rather than contributing it to a machine room resource pool. Unfortunately, the office may accessible by many people from facilities management staff to anyone who picks the office lock. A tamper-resistant device would be safer in an environment than a device without any physical protection. A second example is security of a device in transit between locations.

If the device does not provide for its own physical security, an adversary could tamper with the device while it is being shipped between locations.

Over the last decade, there have been a variety of projects that investigated the issue of physically secure processing. The μAbyss [Weingart87, White87] and Citadel [White91] projects built physically secure boards for PCs that provided an environment for secure computation. These boards allow some portion of an application to run in an environment that could authenticate to a distributed application, because the boards had an identity and cryptographic keys, and perform processing unobserved by an adversary. In his thesis, Bennet Yee explores how to use secure coprocessors to build secure distributed systems in a variety of different applications [Yee94]. All of this earlier work has demonstrated that small tamper-resistant processing cores can serve as the basis for complex and secure distributed applications. Later in this chapter, I will build on the basic idea of tamper-resistant processing and show it can be used to build a secure NASD without requiring the entire device be secure.

In recent years, several companies have introduced secure computing devices. IBM built a product, the IBM 4758, which is directly descended from the Citadel and μAbyss projects and costs $2,000 in single quantities [IBM99]. At the other end of the spectrum are low-cost low-computational power smart cards such as the Schlumberger Cyberflex which costs $16 [Schlumberger99][1]. Compared to smart cards, secure coprocessors normally have much higher degrees of tamper resistance and they are significantly more expensive. In a NASD environment, a filemanager could be kept in an insecure facility if a secure coprocessor was used to protect its cryptographic keys and generate access credentials.

When discussing tamper resistance, the standard rubric for physical security is the Federal Information Processing Standard 140-1[2] (FIPS-140-1), which defines four levels of protection for cryptographic devices [FIPS140-1]. The FIPS criteria also characterizes other attributes of the system, but the physical security is the aspect that I address in this section. Levels one and two provide very little protection of a secure device thus are not relevant to NASD. For physical security, a NASD drive could achieve level three by simply packaging the processor, central ASIC, key memory, encryption, a message authentication code, and key management logic in a hard opaque tamper-evident coating. Level three only requires that the device be tamper-evident, i.e. an inspection of the device will reveal it was tampered with, and makes it more difficult for an attacker to extract secrets. This level of protection would probably stop a casual attacker but not a determined technically adept attacker who didn't care about after-the-fact detection.

FIPS level four requires the device to play an active role in protecting its secrets. Level four requires tamper-response circuitry which *zeroize,* irretrievably erase, security

1. This is a high-end smart card that can implement a wide variety of algorithms. There are cheaper smart cards available which are little more than portable memory devices intended to replace magnetic stripe cards.
2. FIPS 140-1 is currently in a state of review and public comments are being accepted. Reports are that the criteria will be updated and strengthened to include more recent security attacks such as timing and power [Tygar99]

critical state when tampering is detected. A battery backed RAM is normally used to store security critical information because it is the easiest technology to quickly zeroize. When tampering is detected, swift action must be taken to prevent RAM from being imprinted on memory [Gutman96]. The "crow-bar" technique, which shorts the power to ground, is an effective method of high-speed erasing of RAM [Smith98].

Level four also requires the device to zeroize if environmental values, specifically temperature and voltage levels, move outside of an acceptable range because extreme conditions can be used to attack a secure device. The temperature floor prevents an adversary from quickly cooling the device to stabilize RAM [Gutman96]. Voltage spikes can also have a similar effect and imprint RAM with a value. Once imprinted, the attacker can break through the physical barriers and recover the secrets from the RAM even if the tamper-detection circuitry attempts to erase the data. Under normal conditions, the RAM will be imprinted with a value stored for any length of time so the software should repeatedly invert the values to prevent the keys from becoming imprinted in RAM. The environmental requirements are also intended to prevent an adversary from using extreme environments to induce faults in a secure device that may cause the device to release secret information.

The downside of tamper resistance is its additional cost. The sensors required for the environmental protection (thermoresistors, op-amps, resistors) in a level four device are inexpensive standard components. However, a special-purpose physical barrier that can prevent or detect a wide variety of physical attacks is very expensive to manufacture and it uses proprietary technology. Steve Weingart, a leading tamper-resistance expert at IBM, suggests that the cost of the physical barrier is one of the biggest components of current tamper-resistant devices and, with proper interest from material science and chemical engineering communities, more cost-effective and efficient solutions could be developed within five years [Weingart99].

Recent work by Anderson and Kuhn has demonstrated a variety of low-cost attacks against supposedly secure smart card devices that must be accounted for in any implementation [Anderson96b, Anderson97]. They have successfully applied low-cost attacks to tamper-resistant devices such as smart cards and set top boxes. Additionally, they have employed microprobe technology, from cellular biology's tool chest, and dry etching techniques to extract information from tamper-resistant devices. They have shown that transient glitches in power or clock signals are frequently missed by tamper detection circuitry. These glitches can be used in a variety of differential fault analyses [Biham97]. In differential fault analysis, the adversary causes some loosely controlled faulty behavior in a device, which enables the adversary to extract secrets from the device. These recent advances in low-cost attack technology demonstrate that NASD require careful design and active defense to counter even a low-budget educated adversary.

Paul Kocher et al. at Cryptography Research have shown that both timing [Kocher96] and power consumption [Kocher98] leak enough information for an attacker to recover the key given a few thousand samples. When the amount of time for operations changes as a result of key values, an attacker can extract key information from

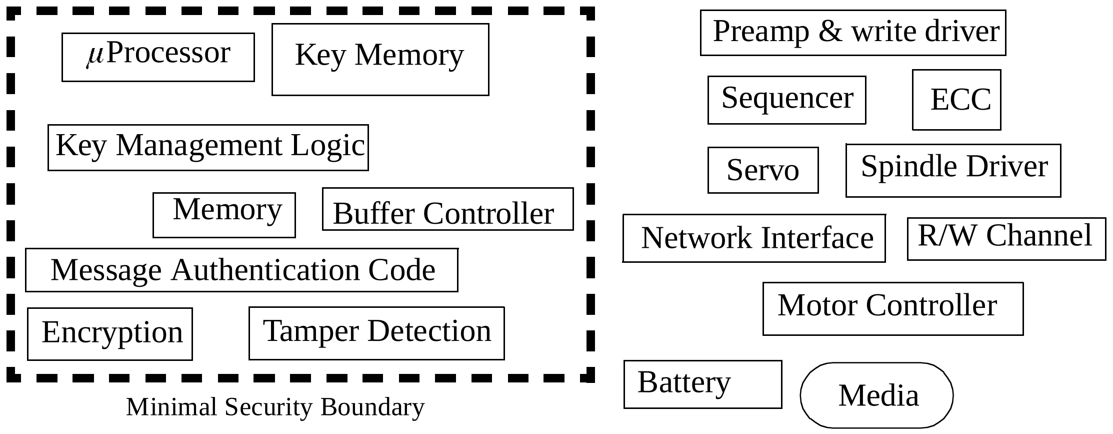


Figure 8-1 Minimal Security Boundary

The elements of the figure within the dotted frame are the security-critical components of a modern disk drive. They are the processing and defensive portion of the NASD, and they are also the minimum set of components that must be protected from tampering in order for keys and key-dependent computation to be kept private. Components outside of the dotted frame are not security critical and can be viewed more as communication mechanisms rather than processing elements.

the amount of time encryption operations take. Kocher suggests that this risk can be reduced by either making all operations constant time, thus sacrificing performance, or introducing enough random noise to mask the signal being leaked. For the information leaked by power consumption, Kocher et al. suggest that introducing noise into the power signal or temporally decorrelating cryptographic operations can help control the rate of information leaked. They also suggest that careful applications of existing cryptographic algorithms or the design of new algorithms may also reduce information leaked through power consumption.

For a tamper-resistant NASD drive, the minimal security boundary, shown in Figure 8-1, includes microprocessor, key memory, encryption, message authentication code, key management logic, buffer controller, tamper-detection circuity, and memory. These functions are where all of the “thought” goes into requests. All key-dependent calculations and defensive mechanisms are contained within this group of functions. A minimum amount of memory must be included within the boundary to provide a workspace for the drive’s security processing function to store data. However, the workspace can be extended by crypto-paging data, using cryptography to protect data stored in insecure memory, to a larger external memory [Yee94].

The functions outside of the minimal boundary are primarily communication channels of the drive rather than logical processing elements. This includes the media, motor controller, R/W channel, preamp and write driver, error-correcting control, sequencer, servo controller, and SCSI interface. The media can be viewed as a communication channel between the secure portion of the system and itself with a very

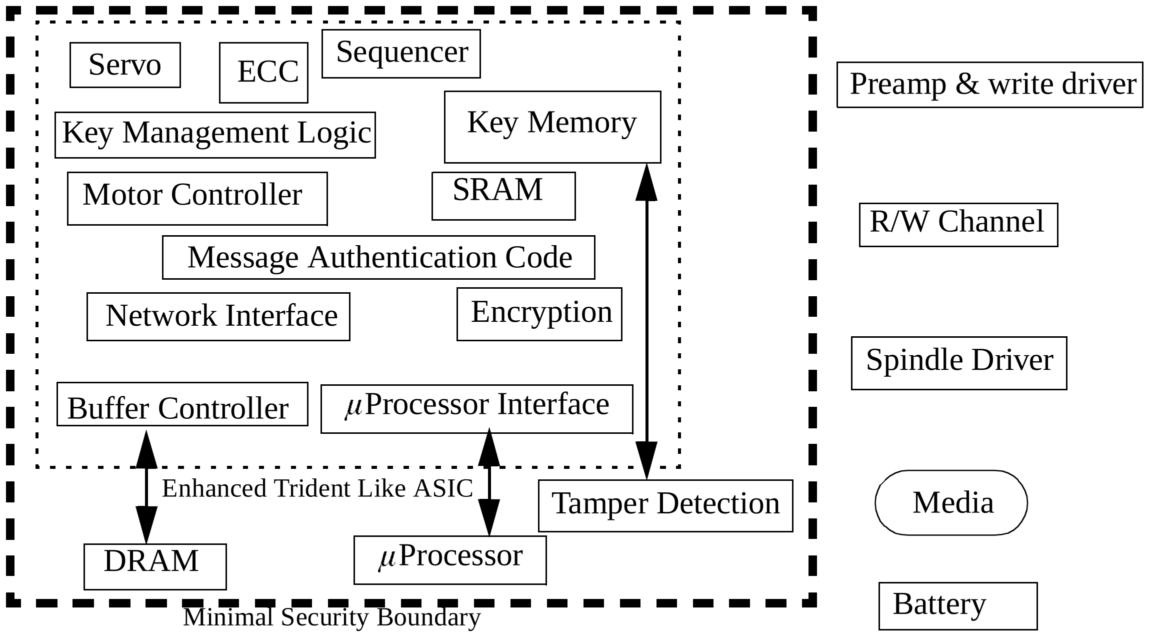


Figure 8-2 Security Boundary in a Viking Like Implementation

The components in the dashed box are a cryptographically-enhanced Trident ASIC, while the components in the dotted box are those that must be physically protected in order for the drive to provide a high degree of assurance to clients. Other combinations of drive functions could be included in the security boundary as long as the minimal set depicted in Figure 8-1 is included. This combination expands on a current disk drive's level of integration but there is a wide range of alternatives.

large latency. If the end points of a message exchange are secure, then they can construct a secure channel despite an insecure communications channel.

To adapt the core drive ASIC of a Quantum Viking, called the Trident (shown earlier in Figure 7-1 on page 144), to a secure version, the DRAM, microprocessor, tamper-detection circuitry, cryptographically-enhanced Trident ASIC must be moved within the security boundary as shown in Figure 8-2. Gibson et al. have suggested that next-generation drive ASICs can integrate the microprocessor and cryptographic support onto a Trident-like ASIC without increasing the die size [Gibson98]. In addition to this functionality, only the tamper-detection circuitry must be integrated into the primary ASIC.

Tamper resistance is not a cure-all for security. Availability, a critical concern for storage devices, remains the same because an adversary can still destroy a storage device or its connections to the network. Availability may even be slightly worse because the tamper-detection circuitry and sensor's failure mode is to zeroize critical secrets which renders the device unusable. An application built on NASD devices with both high availability constraints and security constrains would need to use RAID [Patterson88] or equivalent technologies to build availability above the storage device layer.

Past research has provided a good understanding of how tamper-resistant devices can be built and applied in distributed systems. As a result, in part, of the work in the research community, vendors have developed a variety of tamper-resistant devices. These same technologies can be applied to a NASD device to reduce the physical security assumption down to the tamper resistance of the drive's core processing. This enables the drive to participate as part of a secure distributed application independent of its physical location.

8.3 Media Cryptography

Once we have some degree of physical security, we can provide increased data security. The drive can encrypt data in the tamper-resistant core before it is stored on the media. Applications are free to store encrypted data at the drive but encrypting the data at the storage device enables better control of information. Additionally, placing encryption support at the device level enables more efficient encryption through parallelism; otherwise while the network may be safe, backups are unsafe.

Regardless of the physical security of a drive, an application can always store data on a NASD in encrypted form, although this provides weaker security than if the device handled encryption. The disadvantage of application-level encryption is that all the data must be read and rewritten if someone's access is revoked. The revokee may have read and cached data; rewriting it at this point will not change what the revokee knows. However, rewriting data will prevent future accesses to data which the revokee was entitled to access but did not cache. Similarly, two clients with the appropriate read keys for a file can observe exactly what the other is reading. By encrypting data as it goes over the network as described in Section 4.4.4.2, the drive and client prevent adversaries from learning what data is being read despite the fact that the adversary may be allowed to read the same data. By encrypting at the media level and securely protecting the keys, the drive prevents an adversary who gains physical access to the drive from also gaining access to all the data. Good examples of application layer encryption and a discussion of many of the relevant issues are Matt Blaze's Cryptographic File System [Blaze93] and the Transparent Cryptographic Filesystem Project at Universita di Saleron [Cattaneo99]. Jim Hughes at StorageTek corporation is also working on a similar system called SFS which is intended for distributed systems [Hughes98].

If data is stored and encrypted by the drive then an adversary with physical access to the data will be unable to recover the data unless she can somehow acquire the necessary cryptographic keys. If a drive uses active tamper resistance to protect keys, then an adversary who attempts to breach the physical security will trip a sensor which will zeroize the keys and effectively "erase" all the data. Since all the data is stored on the media encrypted under a key which no longer exists, an adversary will need to break the encryption system to retrieve any data.

A drive should use a value derived from one of the long-term keys, such as the master or drive key, to protect stored data. A long-term key should be used because all data must be read, decrypted, re-encrypted, and rewritten whenever the media data encryption key changes so this should be an infrequent event. In order to limit the amount of data encrypted with a single key available to an attacker and complicate an attack even when the adversary can read the media, the long-term key should be combined with other information, such as the disk block ID or object ID, in a one-way function, similar to the technique employed in Section 4.3.2.2, to produce a unique media encryption key for each disk block or object rather than using a single key for all stored data.

Without tamper resistance, the drive can't keep a secret. However, the drive provides an opportunity to increase the performance for an application-level security function, which is similar to the more generic performance improvements from moving application functions to the storage device [Riedel98a]. The NASD API could be extended to include a client-specified MAC or encryption key on each operation that the drive uses to process the request separately from the cryptography used for communications. For a client operating in a secure environment, this allows the client to *not* perform cryptographic operations, since communications are assumed secure, and still have data stored with an application-layer MAC or encrypted. By slightly extending the NASD API, I have allowed clients to use the parallelism of a large number of storage devices for function which logically occur at the application level. Once the request is completed, the drive discards the keys so the data can only be retrieved or verified with application level keys. The data can safely be backed up and an adversary will neither be able to read the data, if it is encrypted, nor modify the data, if it is MAC'd.

A tamper-resistant drive can prevent an adversary with access to the drive from modifying stored data and remaining undetected. The drive uses the secrets protected in the tamper-resistant core to generate a MAC of each disk block stored on the media. Just as a MAC protects message from being modified by an attacker, a MAC can also protect data on the media since the data is essentially a message from the drive and to the drive.

8.4 Chapter Summary

Tamper resistance offers an appealing alternative to continuously staffed and tightly watched machine rooms. By reducing the security assumption to the tamper-resistance of a device from the security of a machine room, it is easier to make assertions about the security of the device.

Low levels, FIPS level 3 or less, of tamper resistance can be easily added to a NASD to prevent the release of cryptographic keys and protect computations. FIPS level 4 is possible although extremely expensive. Once a drive can be trusted to keep a secret, it can encrypt data stored on the media which prevents an adversary from reading the data without compromising the tamper resistance.

Chapter 9: Conclusion and Future Work

Network attached storage is already moving from a research vision and into production. The synergy of several technology trends such as I/O bound applications, new drive attachment technologies, rapidly increasing drive performance, convergence of peripheral and interprocessor switch networks, and an excess of on-drive transistors is making network attached storage a compelling architecture for high performance commodity storage subsystems. When storage is promoted to a first-class network entity, it is exposed to direct attacks over the network. At a high level, this dissertation describes and analyzes a solution to network attached storage's security problem.

Network attached storage's security system must enable an application to control access to remote storage in addition to protecting the integrity and/or privacy of data. These security goals must be achieved without significantly diminishing the performance advantages of network attached storage.While we would prefer all users to be benevolent and computing environments to be safe, most environments have malicious adversaries, both internal and external to an organization, who attempt to violate the application's security policy by reading, modifying, reordering, deleting or replaying network packets. These are the types of attacks that NASD's security system addresses.

I present a cryptographic capability security system which is general enough for application specific filemanagers to efficiently enforce most security policies over their storage. The security of the entire distributed system depends on assumptions about the strength of the underlying cryptographic tools: cryptographic message digests, message authentication codes, and encryption (only necessary to provide privacy), in addition to the proper protection of cryptographic keys. This system enables filemanagers to asynchronously make policy decisions that are synchronously enforced by the storage devices on every operation. Further, the drive and client cooperate in protecting the integrity and/or privacy of operations to meet filemanager requirements. The security analysis of this design and the asynchronous involvement of the filemanager justify the first part of my thesis statement:

> ***A cryptographic capability system designed for a range of distributed storage applications provides fundamental scalability because it enables reuse of policy decisions and unmoderated, parallel interactions between application and device.***

Throughout the dissertation, I refined the basic security system design to increase both performance (by evolving the basic design and adding hardware support) and security (by adding tamper resistance). By moving from a capability system to a system based on remote execution techniques, I reduced the number of client requests to the filemanager which increases the filemanager's scalability. Additionally, moving more function into the drive enabled dynamic dependency checking and multiple object operations; features that capabilities do not provide. The precomputation optimization reduces the amount of computation necessary to protect integrity on read operations so a given set of hardware can deliver more integrity protected bandwidth. In the CMU prototype, which is limited by cryptographic bandwidth, these optimizations improve integrity protected read bandwidth by a factor of 5 on large reads. Additionally, there is a wide variety of cryptographic ASICs and logic cores which can be used to improve system performance at a small cost. If the drive hardware can perform message digests at less than full duplex network data rates but still deliver more than 1/3 of the network duplex data rate, then the clients will see a less than 10% increase in latency on an average filesystem request. The precomputation optimization experiments along with simulation of hardware requirements support the second of my thesis claims:

> ***Commodity storage devices can be designed to inexpensively provide security and high bandwidth.***

Furthermore, tamper-resistant hardware can provide stronger security guarantees without requiring a secure facility for the storage devices.

Together, the preceding research makes the following contributions:

- An argument for the separation of policy and mechanism in a commodity network attached storage system enforced by a cryptographic capability system,
- The basic design and implementation of a security system for network attached storage, based on cryptographic capabilities,
- An understanding of the scalability advantages of aggregation mechanisms that move more functionality to the storage device,
- A proposal to use precomputed hash values as the basis for a new message authentication code structure,
- A demonstration of the performance advantage of the new message authentication code structure,
- An understanding of the performance requirements for message authentication code computation,
- An evaluation of available options for hardware support, and
- A high-level sketch of a NASD design based on tamper-resistant hardware.

This work describes the challenges that high-performance network attached storage poses for security. I have presented a solution targeted at the commodity end of the potential solutions emphasizing the low cost and limited capabilities of the storage device.

If these ideas are adopted and standardized across storage devices, secure commodity network attached storage devices will be available for high-performance storage systems that deliver greater throughput and scalability than current technology.

9.1 Future Work

As part of this research, I have surveyed the current commercially available cryptographic chips and logic cores and argued that these components could be successfully integrated into a storage device's internal architecture. An obvious piece of future work is to implement such a device. By integrating the security into the drive ASIC, I could better understand the cost and performance implication of integrating cryptography into a commodity embedded processing device.

Further, the simple basic capability system is designed to minimize the amount of work that the storage device must perform. However, I have not evaluated the difficulty or cost of implementing this functionality in a special purpose ASIC on the drive rather than its processor. Storage devices have many fast-path optimizations for common case operations or operations that can be serviced quickly, such as cache reads. Pushing the capability system a step further and integrating it into the primary ASIC, perhaps in conjunction with integrating the microprocessor, is the next step in validating the assumption that a relatively simple security system is well suited for a high-performance low-cost storage device.

Similarly, it would be useful to understand how the NASD ideas and NASD security system can be implemented in a RAID controller. For example, early NASD devices may be network attached RAID controllers sitting in front of an array of SCSI disks. At a high level, ignoring fault tolerance issues, this is no different from a very large and very fast disk. However, the internal architecture and the cost concerns are quite different between a disk drive and a RAID controller. While both are high-performance, highly-optimized devices, the RAID controller already has more functionality implemented than a simple storage device and is less sensitive to cost. The most thorough way to investigate this difference is to implement the NASD interface on an existing RAID controller.

While this thesis focuses on filesystems, it also explores some of the challenges of implementing a database system on a network attached storage architecture. The dissertation is limited by a lack of good database workloads and descriptions of their behavior with respect to security policies. With better information on databases, both the basic capability design and the remote execution based alternative could be analyzed for these workloads and refined, along with the basic NASD interface, to better meet the needs of database systems.

In Chapter 3, I discussed some of the implications of database security on NASD systems. The simplicity of the capability system limits NASDs ability to handle to complex data dependent access control decisions that a DBMS can make. However, the remote execution solutions presented in Chapter 3 may be also make NASD into a better target for a DBMS system with a rich security policy. By allowing the application to place more of its functionality at the storage device, the application is empowered to make the more complicated and potentially expensive access control needed for some databases.

My research has been based partially on traces collected from AFS and NFS filesystems running in academic environments. Further studies including traces from non-academic research environments as well as non-Unix based filesystems, such as CIFS, would help strengthen the conclusions or identity the biases introduced by the workloads.

Similarly, it is important to understand how NASD systems will differ from SAD systems. My results are based on a study of existing SAD systems and ports of SAD systems to the NASD architecture. While I expect these are a good indicator of future NASD systems, it is not clear that a system specifically developed to run on NASD, i.e. a native NASD application, might behave somewhat differently or pose additional requirements. Currently, there is an effort to develop a native NASD high-performance filesystem within the Parallel Data Lab but it is still an open question of how its access patterns and behavior will differ from the systems studied. Additionally, the workloads I have studied are server workloads which I use to approximate a storage device’s workload. A native application running on prototype NASDs will generate a true NASD workload which can be contrasted to the server workload approximations.

Over the years of research culminating in this dissertation, I’ve seen the rapid growth of both the internet and networked devices. You can now purchase a variety of devices, such as cameras, disks, and displays, that plug into a network. I see a need for a general purpose solution to security and access control for arbitrary commodity devices that are simply plugged into the network with little or no configuration. Because these are commodity devices, they share some qualities with NASDs. They all have relatively simple interfaces and are designed with cost as an important factor. Using an asynchronous control system, such as the one I have proposed for NASD, for other types of devices would be useful for controlling the large array of network devices that will exist in future homes and offices without requiring that each device be directly connected to a server. However, the server must be able to define a policy that meets its specific needs and the storage devices must enforce this policy over every operation which the NASD architecture will enable.

Appendix A: GNY Analysis

Formal analysis techniques are tools to understand both the strengths and the limitations of cryptographic protocols. Over the years, a wide variety of techniques have been developed (a good survey of these techniques can be found in [Meadows95]). The Burrows-Abadi-Needham (BAN) [Burrows90] logic is one of the most well known and widely used analysis techniques due to its simplicity and utility. It is an example of a class of techniques called logics of belief. A logic of belief allows you to reason about what beliefs the principals, the parties involved in the protocol, should hold at different points in the protocol.

A BAN analysis produces a list of logical steps that allow principals to rationally hold a set of goal beliefs at the conclusion of the protocol and a set of initial beliefs that the principals must hold in order for the steps to be applicable. The first step is to *idealize* the protocol into a formal description which abstracts or eliminates some details in order that the rules of the logic can be applied. The difference between the idealized version and implemented version of a protocol is one of the most often cited criticisms of BAN-like analysis techniques because the subtle nuances in the difference can allow a protocol to be proved correct despite having serious flaws. However, when carefully done, the BAN logic is a useful tool for understanding a cryptographic protocol.

All principals have an initial set of beliefs, in the case of NASD the principals are drives, clients, and filemanagers. When a principal receives a message, the principal can derive an additional set of beliefs based on the logic’s postulates. At the completion of the protocol run, the principals should be able to arrive at a pre-determined conclusion. A failure to reach the desired conclusion indicates a flaw in the protocol or that additional assumptions are necessary.

As part of the idealization step, the BAN technique disregards all unencrypted messages as “hints” that have no impact on the outcome of the protocol. A consequence of this is that BAN is difficult to apply to NASD because NASD makes extensive use of plaintext messages. These plaintext messages are significant to the protocol because they are used as inputs to a message authentication code algorithm which both protects integrity and generates cryptographic keys. Since BAN does not include message authentication codes, BAN can discard plaintext messages but this makes it an inappropriate tool for analyzing NASD.

It is not necessary to look far beyond BAN to find a logic of belief that includes message authentication codes. The Gong-Needham-Yahalom (GNY) [Gong90] logic, a close relative of BAN, includes support for a keying hash function, essentially message authentication codes. GNY normally denotes the use of hash functions with a secret key *S* as *H(<S>,X)* which I will replace with $MAC_S(X)$ in order to be clearer and more consistent with how I have described the NASD protocol. GNY also separates the notion of *believing* a formula to be true and *possessing* a formula. This feature helps capture the implicit communication of the private portion of the access credential without requiring that the idealization make the communication explicit.

In this appendix, I use GNY to analyze the NASD protocol being used to protect the integrity of an entire request. In order to analyze the NASD protocol, I also include a very simple client-filemanager protocol to illustrate one way to achieve the necessary beliefs for the client and drive to interact. The analysis illuminates the specific assumptions that must be true for the protocol to be successful and correct. This includes assumptions about timestamps and NASD's implicit communication of keys. The analysis clearly demonstrates the points where NASD reaches beyond scope of the analysis technique and where assumptions should be examined in greater detail.

In the analysis, I will frequently refer to the logical postulates of GNY which are fully documented in [Gong90]. In Section 9.2, I briefly introduce notation of GNY and explain how I handle the anonymous clients, since clients are identified only by their access rights, in the NASD analysis. In Section 9.4, I define the logical GNY postulates that I use the analysis.

9.2 Notation

9.2.1 GNY Notation

Briefly, I introduce the notation used in GNY and a more detailed description with examples is included in [Gong90]. GNY reasons about formulas, bit-strings with a particular value in a protocol run, and principals, the communicating parties. Let X and Y range over formulas and P and Q range over principals. GNY uses the following statements:

- $P \triangleleft X$: *P is told X*. P either explicitly receives X or can directly compute X from a message.
- $P \ni X$: *P possesses X*. P knows the value of X and also the value of anything computable from X.
- $P \mid\sim X$: *P once conveyed* X. At some point in the past, P said X.
- $P \mid\equiv \#(X)$: *P believes X is fresh*. P believes, or is entitled to believe, that X has not been used for the same purpose at any time before the current protocol run.

- $P \mid\equiv \phi(X)$: *P believes that X is recognizable.* P would recognize a valid value of *X* because of expected characteristics such as the structure of a public credential, the structure of a timestamp, or redundancy in *X*. Primarily, this asserts that an adversary can't replace *X* with random garbage and have it go unnoticed.
- $P \mid\equiv P \overset{K}{\longleftrightarrow} Q$: *P believes K is a suitable secret for P and Q*. *P* believes that *K* is a secret shared between *P* and *Q*. This means that *K* can be used as a secret for encrypting or generating MAC's of messages.
- $*$: This specifies that the statement was not generated by the receiver in this run of the protocol.
- $P \mid\equiv Q \Rightarrow C$: *P believes Q has jurisdiction over C*. *P* believes *Q* is an authority on *C* and should be trusted on this matter.
- $\{X\}_K$: *X encrypted under key K.* A principal who has key *K* will be able to both see and possess *X*.
- $X \rightsquigarrow Y$: *Y is an extension of X*. In this message, *Y* is a precondition on the sender having sent *X*. This is an expression of the belief of the sender and captures the protocol requirements that a sender only send a message *X* if she holds a belief *Y.*

9.3 Anonymous Clients

GNY and other BAN-like formalisms handle named principals. In NASD, the clients are anonymous from the perspective of the drive, a drive only knows that a request came from someone holding a specific set of access rights, described by their public access credential which was discussed in Section 4.2.2, and the drive is not aware of the user's true identity. I model the public credential, the description of access rights, as a public function *R* of an object identifier *I* which captures the fact that anyone can generate a public access credentials. Anyone can generate a public access credential describing a specific set of rights, system security is a result of the ability of only authorized entities to generate the associated private access credential. A client operating with a specific set of rights is denoted as $\mathrm{Cl}_{\mathrm{R(I)}}$ which acts as a name within the analysis. By overloading the naming of clients, a drive can reason that a request came from a client with a specific set of rights without reasoning about client identities. If a client accesses a drive with different access credentials, the drive will perceive this as multiple unique clients.

9.4 Logical Postulates Used

These are the exact postulates taken from [Gong90] and are included as a reference for the reader.

9.4.1 Being Told Rules

9.4.1.1 T1

$$\frac{P \triangleleft * X}{P \triangleleft X}$$

Being told a “not-originated-here” formula is a special case of being told a formula.

9.4.1.2 T2

$$\frac{P \triangleleft (X, Y)}{P \triangleleft X}$$

Being told a formula implies being told each of its concatenated components.

9.4.1.3 T3

$$\frac{P \triangleleft \{X\}_K, P \ni K}{P \triangleleft X}$$

If a principal is told a formula encrypted with a key he possesses then he is considered to have also been told the decrypted contents of that formula.

9.4.2 Possession Rules

9.4.2.1 P1

$$\frac{P \triangleleft X}{P \ni X}$$

A principal is capable of possessing anything he is told.

9.4.2.2 P2

$$\frac{P \ni X,\ P \ni Y}{P \ni (X,Y),\ P \ni F(X,\ Y)}$$

If a principal possesses two formulae then he is capable of possessing the formula constructed by concatenating the two formulate, as well as a computationally feasible function *F* of them.

9.4.2.3 P3

$$\frac{P \ni X}{P \ni H(X)}$$

If a principal possess a formula then he is capable of possessing a one-way computationally feasible function of that formula.

9.4.3 Freshness Rules

9.4.3.1 F1

$$\frac{P \mid\equiv \#(X)}{P \mid\equiv \#(X,Y),\ P \mid\equiv \#(F(X))}$$

If *P* believes a formula *X* is fresh, then he is entitled to believe that any formula of which *X* is a component is fresh, and that a computationally feasible one-to-one function *F* of *X* is fresh.

9.4.4 Message Interpretation Rules

9.4.4.1 I1

$$\frac{P \lhd\!\!* \{X\}_K,\ P \ni K,\ P \mid\equiv P \xleftrightarrow{K} Q,\ P \mid\equiv \phi(X),\ P \mid\equiv \#(X, K)}{P \mid\equiv (Q \mid\sim X),\ P \mid\equiv (Q \mid\sim \{X\}_K),\ P \mid\equiv Q \ni K}$$

Suppose that for a principal *P* all of the following conditions hold: (1) *P* receives a formula consisting of *X* encrypted with a key *K* and marked with a not-originated-here mark; (2) *P* possesses *K*; (3) *P* believes *K* is a suitable secret for himself and *Q*; (4) *P* believes formula *X* is recognizable; (5) *P* believes that *K* is fresh or that *X* is fresh.

Then *P* is entitled to believe that (1) *Q* once conveyed *X*; (2) *Q* once conveyed the formula *X* encrypted with *K*; (3) *Q* possesses *K*.

9.4.4.2 I3

$$\frac{P \lhd\!\!* H(X,S),\ P \ni (X,S),\ P \mid\equiv P \overset{S}{\leftrightarrow} Q,\ P \mid\equiv \#(X,S)}{P \mid\equiv (Q \mid\sim (X,S)),\ P \mid\equiv (Q \mid\sim H(X, S))}$$

Suppose that for a principal *P* all of the following conditions hold: (1) *P* receives a formula consisting of a one-way function of *X* and *S* marked with a not-originated-here mark; (2) *P* possesses *S* and *X*; (3) *P* believes *S* is a suitable secret for himself and *Q*; (4) *P* believes that either *S* or *X* is fresh.

Then *P* is entitled to believe that (1) *Q* once conveyed the formula *X* concatenated with *S*; (2) *Q* once conveyed the one-way function of *X* concatenated with *S*.

Note: In the NASD analysis H(X,S) is presented as $MAC_S(X)$.

9.4.5 Jurisdiction Rules

9.4.5.1 J1

$$\frac{P \mid\equiv Q \Rightarrow C,\ P \mid\equiv Q \mid\equiv C}{P \mid\equiv C}$$

If *P* believes that *Q* is an authority on some statement *C* and that *Q* believes *C*, then *P* ought to believe in *C* as well.

9.4.5.2 J2

$$\frac{P \mid\equiv Q \Rightarrow Q \mid\equiv *,\ P \mid\equiv Q \mid\equiv Q \mid\equiv C}{P \mid\equiv Q \mid\equiv C}$$

If *P* believes that *Q* is honest and competent, and *P* believes that *Q* believes that *Q* believes in *C*, then *P* ought to believe that *Q* believes in *C*.

9.5 Analysis

The principals involved in the protocol:

- Fm filemanager
- Dr drive
- Cl a client interacting with the filemanager
- $\mathrm{Cl}_{\mathrm{R(I)}}$ a client with the set of rights described by R(I)

The following well known functions are used in the protocol:

- F(T) A function of a timestamp that is used to generate a different value in a reply. This allows the requestor to match requests and replies without the risk of an adversary sending back the request *in place of* a reply.
- R(I) A function of an object identifier that produces a description of a set of access rights. This function is used distinguish between clients based on access rights.

The following terms are also used in the protocol:

- I object identifier
- N nonce
- T timestamp
- $\mathrm{K}_{\mathrm{A,B}}$ a key shared between A and B

The protocol is:

1. $Cl \rightarrow Fm : Cl, I, N$
2. $Fm \rightarrow Cl : \{R(I), MAC_K(R(I)), N\}_{K_{FmCl}}$
3. $Cl \rightarrow Dr : R(I), Request, T, MAC_{MAC_K\langle R(I)\rangle}\langle Request, T\rangle$
4. $Dr \rightarrow Cl : Reply, F\langle T\rangle, MAC_{MAC_K\langle R(I)\rangle}\langle Reply, F\langle T\rangle\rangle$

The first pair of messages are a client obtaining an access credential from the filemanager using an application specific, rather than NASD defined, protocol. This exchange illustrates the *minimal* requirements for the filemanager and client to run a NASD application. In a real system, this message may include application specific mechanisms to authenticate the request from the client. The second pair of messages are the client using the access credential to perform an operation on the drive using the NASD interface.

When analyzing any protocol, the first question is: What is the goal of the protocol? The ideal goal for NASD is for the drive to believe that the filemanager has authorized the specific operation and the client to believe that the drive provided the appropriate reply. However, the concept of "authorization" does not exist in GNY. Staying within the confines of the logic, the goal is for the drive to believe that a client with an appropriate set of rights, $Cl_{R(I)}$, made the request and the client believes that the drive provided the reply. This overloading of the names was described in Section 9.3. Secondly, I want to show that the drive and client believe the messages they received are not replays.

Idealized into GNY-logic, the protocol is:

1. $Fm \lhd *Cl, *I, *N$

2. $Cl \lhd *\left\{R(I) \leadsto Fm \mid\equiv Cl_{R_I} \xleftrightarrow{MAC_K\langle R_I\rangle} Dr, MAC_K(R(I)), N\right\}_{K_{ClFm}}$

3. $Dr \lhd *R(I), *Request, *T, *MAC_{MAC_K\langle R_I\rangle}(Request, T)$

4. $Cl \lhd *Reply, *F\langle T\rangle, *MAC_{MAC_K\langle R(I)\rangle}\langle Reply, F\langle T\rangle\rangle$

The analysis begins with the following assumptions:

$$Cl \ni K_{ClFm} \qquad Cl \mid\equiv Cl \xleftrightarrow{K_{ClFm}} Fm$$
$$Fm \ni K_{ClFm} \qquad Fm \mid\equiv Cl \xleftrightarrow{K_{ClFm}} Fm$$

Both the client and filemanager possess K_{ClFm}. Both the client and filemanager believe they share a valid secret key shared with the other.

$$Dr \ni K_{DrFm} \qquad Dr \mid\equiv Dr \xleftrightarrow{K_{DrFm}} Fm$$
$$Fm \ni K_{DrFm} \qquad Fm \mid\equiv Dr \xleftrightarrow{K_{DrFm}} Fm$$

Both the drive and filemanager possess K_{DrFm}. Both the drive and filemanager believe they share a valid secret key shared with the other.

$$Cl \mid\equiv \#(N) \qquad Cl \mid\equiv \#(T) \qquad Dr \mid\equiv \#(T)$$

The client believes both the nonce and timestamp are fresh while the drive only believes the timestamp is fresh. The drive believes that it can identify a valid timestamp generated by the client. In a run of the protocol, the client and drive will use a protocol specific mechanism, outside of the scope of GNY, to verify these assumptions. If the assumptions fail, the client or drive should abort the protocol run.

$Cl \mid\equiv \phi(R_I)$

The client is able to recognize a valid set of rights. A client can distinguish random garbage from a reasonable set of access rights. This is reasonable since the client specifically requests the rights and the rights have a very specific structure.

$Cl \mid\equiv Fm \Rightarrow Cl_{R_I} \xleftrightarrow{K} Dr$ $\qquad$ $Dr \mid\equiv Fm \Rightarrow Cl_{R_I} \xleftrightarrow{K} Dr$

The client and the drive both believe the filemanager has jurisdiction over quality secrets shared between the drive and any client with a specific set of access rights. These assumptions capture the idea of a filemanager being authorized to issue access credentials.

$Cl \mid\equiv Fm \Rightarrow Fm \mid\equiv *$ $\qquad$ $Dr \mid\equiv Fm \Rightarrow Fm \mid\equiv *$

The client and the drive both believe the filemanager is honest and competent.

Message 1: By T1 and P, I obtain $Fm \ni (Cl, I, N)$. The filemanager now has the proper nonce, *N*, to return in the reply and the proper object identifier, *I*, to generate a description of the access rights that the client will accept.

Message 2: By T1 & T3, the client is able to decrypt the message and obtain it; it is effectively told the message's contents, i.e. $Cl \lhd R(I),\ MAC_K(R(I)), N$.

Applying T2 and P1, the client possesses the contents of the message which include the entire access credential, i.e. $Cl \ni MAC_K(R(I)), R(I)$.

Applying F1, I obtain $Cl \mid\equiv \#(R(I),\ MAC_K(R(I)), N)$. The client believes the message is not a replay. The client knows that this is not an attempt by an adversary to get the client to use an old access credential and generate more MAC'd or encrypted text under an out-of-date access credential key.

Applying I1, I obtain $Cl \mid\equiv Fm \mid\sim R(I),\ MAC_K(R(I)), N$. And applying I7, I obtain $Cl \mid\equiv Fm \mid\sim R(I)$. The client believes the filemanager once conveyed this set of rights.

Applying J2, I obtain $Cl \mid\equiv Fm \mid\equiv Cl_{R_I} \xleftrightarrow{MAC_K\langle R_I\rangle} Dr$. Since a precondition of sending message 2 is that the filemanager believe that the MAC is a valid key, when a client receives a fresh message 2, the client assumes the filemanager believes the MAC is a valid key. Applying J1, I obtain $Cl \mid\equiv Cl_{R_I} \xleftrightarrow{MAC_K\langle R_I\rangle} Dr$. The client now has a key that it can use to act as $\mathrm{Cl}_{\mathrm{R(I)}}$.

Message 3: Applying T1 and P1, I obtain $Dr \ni R(I), Request, T, MAC_{MAC_K\langle R(I)\rangle}\langle Request, T\rangle$. The drive possesses the entire request.

By P2 & P4, I obtain $Dr \ni MAC_K(R(I))$. The drive has the appropriate key.

At this point, the analysis steps beyond the postulates in logics of belief like GNY. GNY does not have a postulate to describe the non-standard belief in the use of a MAC for access key derivation, which was discussed in Section 4.2.1. In order for the analysis to continue, I formalize the drive's belief in key derivation as:

$$Dr \mid\equiv Fm \mid\equiv \forall I, Cl_{R(I)} \xleftrightarrow{MAC_K(R(I))} Dr$$

The drive believes that the file manager believes that for any NASD ID the *$MAC_K(R(I))$* is a valid key. Now, by J1, I obtain $Dr \mid\equiv Cl_{R(I)} \xleftrightarrow{MAC_K(R(I))} Dr$. The drive believes that the MAC is a valid key for a request from a client with rights *R(I)*.

By I3 & I7, I obtain $Dr \mid\equiv Cl_{R(I)} \mid\sim Request$. The drive believes the client once made the request.

By F1, I obtain $Dr \mid\equiv \#(Request, T)$. The request is not a replay.

The drive now concludes that the request is both valid and fresh so should be processed. By valid, I mean the drive believes the request came from a client with the rights described by *R(I)*. Outside of the scope of the GNY analysis, the drive must determine if *R(I)* enables *Request*. This check involves checking several NASD-specific attributes of the request and rights description including the object IDs, byte-ranges, access control version numbers, and list of allowd operations. If these checks fail, the drive will halt the protocol run.

Message 4: By F1, I obtain $Cl \mid\equiv \#(Reply, F(T))$. The reply is not a replay.

By T1 and P1, I obtain $Cl \ni (Reply, F(T))$. The client possesses the message.

By I3 and I7, I obtain $Cl \mid\equiv Dr \mid\sim Reply$. The client believes the reply came from the drive.

Based on an initial set of assumptions, I have shown why the drive is able to conclude that the request came from a client with the specified rights. I have also shown why the client can conclude the reply came from the drive. Both the client and drive can conclude that the messages were fresh. As part of the analysis, I have formalized the drive's belief in the validity of the MAC as a key generator which is a necessary step outside of the basic postulates of GNY.

References

[Amiri99]	Amiri , K., Petrou, D., Ganger, G., and Gibson, G., *Dynamic Function Placement in Active Storage Clusters*, School of Computer Science, Carnegie Mellon University, Technical Report, CMU-CS-99-140, June 1999.
[Anderson98]	Anderson, D., *Network Attached Storage Resarch*, `www.nsic.org/nasd/meetings.html`, March 1998.
[Anderson96a]	Anderson, R. and Biham, E."Tiger: A Fast New Hash Function," *Proceedings of the Third International Workshop on Fast Software Encryption, Lecture Notes in Computer Science Vol. 1039*, Springer-Verlag, 1996.
[Anderson96b]	Anderson, R., Kuhn, M., "Tamper Resistance -- a Cautionary Note," *Proeedings of the Second USENIX Workshop on Electronic Commerce*, November 1996, Oakland, CA, pp. 1-11.
[Anderson96c]	Anderson, T., Dahlin M., Neefe, J., Patterson, D., Roselli, D., and Wang, R., "Serverless Network File Systems," *ACM Transactions on Computer Systems*, Vol.14, No.1, February 1996.
[Anderson97]	Anderson, R., and Kuhn, M., "Low Cost Attacks on Tamper Resistant Devices," *Security Protocol Workshop '97*, April 1997, Paris, France.
[ANSI85]	Amercan National Standards Institute, "Financial institution message authentication (wholesale)," *ANSI X9.17-1985*, American Bankers Association, 1985.
[ANSI86]	American National Standards Institute, "Small Computer Systems Interface (SCSI) Specification," *ANSI X3.131-1986*, 1986.
[Apple98]	Apple Computer, "AppleTalk Filing Protocol Version 2.1 and 2.2," *AppleShare IP 6.0 Developers Kit*, 1998.
[Arnold98]	Arnold, J. M., "Mapping the MD5 Hash Algorithm onto the NAPA Architecture", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1998, pp. 267-268.

[Asic99] Asic International, *Ai-SHA-1 Hash Function Core,* `www.asicint.com/cores/sha-1.htm`, January 1999.

[Baker91] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., Ousterhout, J. K., "Measurements of a Distributed File System," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991, pp. 198-212.

[Bellare94] Bellare, M., Goldreich, O., and Goldwasser, S., "Incremental cryptography: the case of hashing and signing," *Advances in Cryptology (CRYPTO 94*), *Lecture Notes in Computer Science Vol. 839,* Springer-Verlag, 1994.

[Bellare95] Bellare, M., Guerin, R., and Rogaway, P., "XOR MACs: New methods for message authentication using finite pseudorandom functions". *Advances in Cryptology (CRYPTO 95), Lecture Notes in Computer Science Vol. 963,* Springer-Verlag, 1995.

[Bellare96a] Bellare, M., Canetti, R., and Krawczyk, H., "Keying Hash Functions for Message Authentication," *Advances in Cryptology (CRYPTO 96), Lecture Notes in Computer Science Vol. 1109*, 1996.

[Bellare96b] Bellare, M., Canetti, R., and Krawczyk, H.,"Pseudorandom functions revisited: The cascade construction and its concrete security," Extended abstract in *Proceedings of the 37th Annual Symposium on the Foundations of Computer Science,* IEEE, 1996.

[Bellare97a] Bellare, M., Desai, A., Jokipii, E. and Rogaway, P., "A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation," Extended abstract in *Proceedings of 38th Annual Symposium on Foundations of Computer Science,* IEEE, 1997.

[Bellare97b] Bellare, M., and Micciancio, D., "A New Paradigm for collision-free hashing: Incrementality at reduced cost," *Advances in Cryptology (EUROCRYPT 97)*, *Lecture Notes in Computer Science Vol. 1233*, Springer-Verlag, 1997.

[Benner96] Benner, A. F., *Fibre Channel: Gigabit Communications and I/O for Computer Networks*, McGraw Hill, 1996.

[Berendschot95] Berendschot, A., et al., "Integrity Primitivies for Secure Information Systems. Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040)," *RIPE Integrity Primitives, Lecture Notes in Computer Science Vol. 1007,* Springer-Verlag, 1995.

[Bershad88] Bershad, B., and Pinkerton, C. B., "Watchdogs: Extending the UNIX File System," *Proceedings of the Winter 1988 USENIX Conference*, Dallas, Texas, 1988, pp. 267-275.

[Biham97] Biham, E., and Shamir, A. “Differential Fault Analysis of Secret Key Cryptosystems,” *Advances in Cryptology (CRYPTO* 97), *Lecture Notes in Computer Science Vol. 1294*, 1997, pp. 513-525.

[Birrel80] Birrel, A. D., and Needham, R. M., “A Universal File Server,” *IEEE Transactions on Software Engineering*, September 1980.

[Black99] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P., “UMAC: Fast and Secure Message Authentication,” *Proceedings of CRYPTO 99*, 1999.

[Blaze93] Blaze, M., “A Cryptographic File System for Unix,” *Proceedings of the First ACM Conference on Computer and Communications Security,* Fairfax, VA, ACM Press, November 1993.

[Blaze96] Blaze, M., Diffier, W., Rivest, R., Schneier, B., Shimomura, T., Thompson, E., and Wiener, M., *Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security,* `www.counterpane.com/keylength.html`, January 1996.

[Blum83] Blum, L., Blum, M., and Shub, M., “Comparison of two pseudo-random number genertors,” *Advances in Cryptology (CRYPTO 83),* Plenum Press, New York, pp. 61-78, 1984.

[Boden95] Boden, N.J., et al., Myrinet: A Gigabit-per-Second Local Area Network, *IEEE Micro*, February 1995.

[Budiu99] Budiu, M., and Goldstein, S.C., Fast Compilation for Pipelined Reconfigurable Fabrics, *Proceedings of FPGA ‘99*, 1999.

[Burrows90] Burrows, B., Abadi, M., and Needham, R. M., “A Logic of Authentication.,” *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 18-36, 1990.

[Cabrera91] Cabrera, L., and Long, D., Swift: Using Distributed Disk Striping to Provide High I/O Data Rates, *Computing Systems* 4:4, Fall 1991.

[Callaghan95] Callaghan, B., Palowski, B., and Staubach, P., *NFS Version 3 Protocol Specifcation*, Internet RFC 1813, June 1995.

[Cattaneo99] Cattaneo, G., Persiano, G., *Transparent Cryptographic File System*, tcfs.dia.unisa.it, January 1999.

[Cao93] Cao, P., et al., The TickerTAIP Parallel RAID Architecture, *ACM ISCA*, May 1993.

[Cirrus99] Cirrus Logic, *Datasheet CL-SH8668: 3Ci Integrated ATA Drive Electronics*, `www.cirrus.com/3ci`, March 1999.

[Cox95] Cox, B., Tygar, J.D., Sirbu, M., “NetBill Security and Transaction Protocol,” *Proceedings of the First USENIX Workshop on Electronic Commerce*, July 1995, New York, New York.

[Dahlin94] Dahlin, M. et al., “Cooperative Caching: Using Remote Client Memory to Improve File System Performance,” *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pp. 267-280, November 1994.

[Daley65] Daley, R.C., and Neuman, P.G., “A General Purpose File System for Secondary Storage,” *Proceedings of AFIPS Fall Joint Computer Conference*, Vol. 27, Part 1, Washington, D. C., Spartan Books, 1965, pp. 213-229.

[DEC93a] Digital Equipment Coroporation, *OpenVMS System Manager’s Manual: Essentials*, Maynard, Mass., Digital Equiptment Corporation, May 1993.

[DEC93b] Digital Equipment Corporation, *OpenVMS VAX Guide to System Security*, Maynard, Mass., Digitial Equipment Corporation, May 1993.

[Dennis66] Dennis, J. B., Van Horn, E. C., “Programming Semantics for Multiprogrammed Computations,” *Communications of the ACM*, Vol. 9, No. 3, pp.143-155, March 1966.

[deJonge93] deJonge, W., Kaashoek, M.F., and Hsieh, W.C., “The Logical Disk: A New Approach to Improving File Systems,” *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[Diffie79] Diffie, W., and Hellman, M.E., “Privacy and Authentication: An Introduction to Cryptography,” *Proceedings of the IEEE*, Volume 67, No. 3, March 1979, pp. 397-427.

[Dobbertin96] Dobbertin, H.., “The Status of MD5 After a Recent Attack,” *RSA Labs’ CryptoBytes*, Vol. 2 No. 2, `www.rsa.com/rsalabs/pubs/cryptobytes.html`, Summer 1996.

[DoD85] Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

[Drapeau94] Drapeau, A. L., et al., “RAID-II: A High-Bandwidth Network File Server,” *ACM ISCA*, 1994.

[Eberle92] Eberle, H., “A High-Speed DES Implementation for Network Applications,” *Advances in Cryptology (CRYPTO 92), Lecture Notes in Computer Science Vol. 740*, Springer Verlag, pp. 521-539.

[EFF99] Electronic Frontier Foundation, “RSA Code-Breaking Contest Again Won by Distributed.NET and Electronic Frontier Foundation (EFF),” Press Release, January 19, 1999.

[Finley99] Finley, S., Cognitive Designs, Inc., Personal Communication, January 1999.

[FIPS140-1] Federal Information Processing Standard, FIPS 140-1, *Security Requirements for Cryptographic Modules*, January 1994.

[FIPS180-1] Federal Information Processing Standards, FIPS180-1, *Secure Hash Standard*, April 1995.

[Frisch98] Frisch, A., *Essential Windows NT System Administration*, O'Reilly & Associates, 1998.

[Garfinkle96] Garfinkle, S., and Spafford, G., *Practical UNIX & Internet Security*, O'Reilly & Associates, Inc., 1996.

[Gibson92] Gibson, G., "Redundant Disk Arrays: Reliable, Parallel Secondary Storage," MIT press, 1992.

[Gibson96] Gibson, G., Nagle, D., Amiri, K., Chang, F., Feinberg, E., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., and Rochberg, D., *A Case for Network-Attached Secure Disks*, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-96-142, 1996.

[Gibson97a] Gibson, G., Nagle, D., Amiri, K., Chang, F., Feinberg, E., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., and Zelenka, J., "File Server Scaling with Network-Attached Secure Disks," *Proceedings of the ACM International Conference on Measurement and Modelling of Computer Systems*, Seattle, WA, June 1997.

[Gibson97b] Gibson, G., Nagle, D., Amiri, K., Chang, F., Gobioff, H., Riedel, E., Rochberg, D., and Zelenka, J., *Filesystems for Network-Attached Secure Disks*, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-97-118, 1997.

[Gibson98] Gibson, G., Nagle, D., Amiri, K., Butler, J., Chang, F., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., Zelenka, J. "A Cost-Effective, High-Bandwidth Storage Architecture," *Proceedings of the 8th Conference on Architectual Sypport for Programming Languages and Operating Systems*, 1998.

[Giganet98] Giganet GNN 1000, *GNN 1000 Product Description*, `www.giganet.com/products/GNN1000.html`, 1998.

[Gheorghiu98] Gheorghiu, G., Ryutov, T., and Neuman, B. C., "Authorization for Metacomputing applications," *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[Goldberg96] Goldberg, I., and Wagner, D., "Randomness and the Netscape Browser," *Dr. Dobbs Journal*, January 1996.

[Goldreich86] Goldreich, O., Goldwasser, S., and Micali, S., "How to construct random functions," *Journal of the ACM*, Vol. 33, No. 4, 1986, pp. 210-217.

[Gong90] Gong, L., Needham, R., and Yahalom, R., "Reasoning about Belief in Cryptographic Protocols," *Proceedings of the IEEE Symposium on Resesearch in Security and Privacy*, Oakland, CA, May 1990, pp. 234-248.

[Gong92] Gong, L., "A Security Risk of Depending on Synchronized Clocks," *ACM Operating Systems Review*, Vol. 26, No. 1, pp. 49-53, January 1992.

[Gong93] Gong, L., "Variations on the Themes of Message Freshness and Replay," *Proceedings of the IEEE Computer Security Foundations Workshop VI*, Franconia, New Hampshire, June, 1993, pp. 131-136.

[Gosling96] Gosling, J., Joy, B., and Steele, G., *The Java Language Specification*, Addison-Weseley, 1996.

[Griffioen94] Griffioen, J., and Appletone, R., "Reducing File System Latency Using a Predictive Approach," *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[Grochowski96] Grochowski, E., and Hoyt, R. F., "Future Trends in Hard Disk Drives," *IEEE Transactions on Magnetics*, Vol. 32, No 3., May, 1996.

[Guerro95] Guerrero, F., and Noras, J. M., "Customised Hardware Based on the REDOC III Algorithm for High-Performance Date Ciphering," *Proceedings of the International Workshop on Field-programmable Logic and Applications*, Augsust 1995.

[Gutman96] Gutman, P., "Secure Deletion of Data from Magnetic and Solid-State Memory," *Proceedings of the Sixth USENIX Security Symposium*, July 1996.[1]

[Hartman93] Hartman, J.H., and Ousterhout, J.K., "The Zebra Striped Network File System," *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[HiFn99] Hi/fn, Inc. HiFn, 7751 Encryption Processor Data Sheet, 1999.

[Hitz90] Hitz, D. et al., "Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers," *Proceedings of the Winter 1990 USENIX Conference*, 1990.

[Hitz94] Hitz, D. et al., "File Systems Design for an NFS File Server Appliance," *Proceedings of the Winter 1994 USENIX Conference*, 1994.

1. Peter Gutmann is currently updating this paper to reflect newer technology. The reviser version should be available in mid 1999. Consult his web page for more information: `www.cs.auckland.ac.nz/~pgut001`

[Horst95] Horst, R.W., “TNet: A Reliable System Area Network,” *IEEE Micro*, February 1995.

[Howard88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M., “Scale and Performance in a Distributed File System,” *ACM Transactions on Computer Systems*, Vol. 6, No. 1, Februrary 1988, pp. 51-81.

[Hughes98] Hughes, J., *Information Security, Build Secure File Systems in an Insecure Environment*, Presentation at Carnegie Mellon University’s System Design and Implementation Seminar, November 19, 1998.

[IBM99] IBM Corporation, *IBM 4758 PCI Cryptographic Coprocessor*, `www.ibm.com/security/cryptocards/`, January 1999.

[IBM97] IBM Corporation, *Understanding SRAM Operations*, `www.chips.ibm.com/products/memory/sramoperations/sramop.pdf`, March 1997.

[Intel97] Intel Corporation, *Virtual Interface (VI) Architecture*, `www.viarch.org`, December 1997.

[Jaggar96] Jaggar, D., *ARM Architecture Reference Manual*, Prentice Hall, 1996.

[Jakobson98] Jakobson, M., Shriver, E., Hillyer, B. K., and Juels, A., “A Practical Secure Phsyical Random Bit Generator,” *Proceedings of the 5th ACM Conference on Computer and Communications Security*, November, 1998, San Francisco, pp. 103-111.

[Kaashoek96] Kaashoek, M. F., Engler, D. R., Ganger, G. R., and Wallach, D. A., “Server Operating Systems,” *1996 SIGOPS European Workshop*. Connemara, Ireland, 1996.

[Karger88] Karger, P. A., *Improving Security and Performance for Capability Systems*, University of Cambridge Computer Laboratory Technical Report No. 149, October 1988.

[Katz92] Katz, R. H., “High-Performance Network- and Channel-Attached Storage,” *Proceedings of the IEEE 80:8*, August 1992.

[Kean98] Kean, T. and Duncan, A., “DES Key Breaking, Encryption, and Decryption on the XC6216,” *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998, Napa Valley, California, pp. 310-311.

[Kelsey98] Kelsey, J., Schneier, B, Wagner, D, and Hall, C., “Cryptanalytic Attacks on Pseudorandom Number Generators,” *Proceedings of the Fifth International Workshop on Fast Software Encryption*, March 1998, Sprinter-Verlag, pp. 168-188.

[Kim86] Kim, M. Y., “Synchronized disk interleaving,” *IEEE Transactions on Computers*, C-35, 11 , November 1986.

[Kocher96]	Kocher, P. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” *Avances in Cryptology (CRYPTO 96), Lecture Notes in Computer Science Vol. 1109,* 1996, pp. 104-113.
[Kocher98]	Kocher, P., Jaffe, J., Jun, B., *Introduction to Differential Power Analaysis and Related Attacks,* `www.cryptography.com/dpa/technical/index.html`, 1998.
[Lacy93]	Lacy, J. B., Mitchell, D., and Schell, W. M., “CryptoLib: Cryptogaphy in Software,” *Proceedings of USENIX Security Symposium IV,* USENIX Association, 1993, pp. 237-246.
[Lam92]	Lam, K.Y, and Beth, T., “Timely Authentication in Distributed Systems,” *Proceedings of the European Sympoium on Research in Computer Science, Lecture Notes in Computer Science Vol. 648,* Springer Verlag, 1992, pp. 293-303.
[Lammers99]	Lammers, D., “Cost crunch creates push for single-chip drive,” *EE Times Online,* `www.eet.com/story/industry/semiconductor_news/`OEG1990531S0001, May 29, 1999.
[Leach97a]	Leach, P., *Implementing DFS,* Presentation at Second Common Internet File System Implementor’s Workshop, `www.cifs.com/2ndcifsconf/index.html`, April 1997.
[Leach97b]	Leach, P. J., and Naik, D. C., *A Common Internet File System (CIFS/1.0) Protocol - Internet-Draft Version 1,* `ftp://ietf.org/internet-drafts/draft-leach-cifs-v1-spec-01.txt`, December 1997.
[Lee96]	Lee, E. K., and Thekkath, C.A., “Petal: Distributed Virtual Disks,” *Proceedings of ACM Conference on Architectual Sypport for Programming Languages and Operating Systems,* October 1996.
[Luk97]	Luk, W., Cheung, P. Y. K., and Glesner, M., “A case study of partially evaluated hardware circuits: key-specific DES,” *Proceedings of the 7th International Workshop on Field-programmable Logic and Applications,* September 1997, London, U.K., pp.151-160.
[Madson98]	Madson, C., and Glenn, R., *The Use of HMAC-SHA-1-96 within ESP and AH,* Internet RFC 2404, November 1998.
[Maeda93]	Maeda, C., and Bershad, B., “Protocol Service Decomposition for High-Performance Networking,” *Proceedings of the 14th ACM Symposium on Operating Systems Principles,* December 1993.
[Mann94]	Mann, T., Birrell, A, Hisgen, A., Jerian, C., and Swart, G., “A Coherent Distributed File Cache with Directory Write-Behind,” *ACM Transctions on Computer Systems,* May 1994, pp. 123-164.

[Massigilia94] Massiglia, P., ed., *The RAIDbook*, RAID Advisory Board, 1994.

[Mazieres97] Mazieres, D. *Security and Decentralized Control in the SFS Global File System*, MIT Master's thesis, August 1997.

[McKusick84] McKusick, M.K. et al., "A Fast File System for UNIX" , *ACM Transactions on Computer Systems*, August 1984.

[McCanne93] McCanne, S., and Jacobsen, V., "The BSD Packet Filter: A New Architecture for User-Level Packet Capture.," *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[Meadows95] Meadows, C., "Formal Verification of Cryptographic Protocols: A Survey," *Advances in Cryptology (ASIACRYPT 94), Lecture Notes in Computer Science Vol. 917*, Springer-Verlag, 1995, pp. 133-150.

[Micciancio99] Micciancio, D., Personal Communication, January 21, 1999.

[Microsoft96a] Microsoft Corporation, *Distributed File System: A Logical View of Physical Storage*, White Paper, 1996.

[Microsoft96b] Microsoft Corporation, *Microsoft NT Distributed Security Services: Secure Networking using Windows NT Server Distributed Services Technology Preview*, White Paper, 1996.

[Microsoft98] Microsoft Corporation, *Microsoft Developers Network Web Pages*, msdn.microsoft.com/developer/, 1998.

[Miller88] Miller, S.W., "A Reference Model for Mass Storage Systems," *Advances in Computers 27*, 1988, pp. 157-210.

[Mittra97] Mittra, S., and Woo, T., "A Flow-Based Approach to Datagram Security," *Proceedings of the ACM SIGCOMM '97*, 1997.

[Menezes98] Menezes, A., van Oorschot, P., and Vanstone, S., *Handbook of Applied Crptography*, CRC Press LLC, New York, 1998.

[Mummert94] Mummert, L., and Satyanarayanan, M., *Long Term Distributed File Reference Tracing: Implementation and Experience*, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-94-213, 1994.

[NCSC87] National Computer Security Center, *A Guide to Understanding Audit in Trusted Systems*, NCSC-TG-001 Version 2, Fort Meade, MD, July 1987.

[Necula96] Necula, G., and Lee, P., "Safe Kernel Extensions Without Run-Time Checking," *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Washington, October 1996.

[Nelson88]	Nelson, M.N., Welch, B.B., and Outsterhout, J.K., "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988.
[Neuman93a]	Neuman, B. C. and Stubblebine, S. G., "A Note on the Use of Timestamps as Nonces," *ACM Operating Systems Review*, Vol. 27, No. 2, pp. 10-14, April 1993.
[Neuman93b]	Neuman, B. C., "Proxy-Based Authorization and Accouting for Distributed Systems," *Proceedings of the 13th Intenrational Conference on Distributed Computing Systems*, pp. 283-291, May 1993.
[Neuman94]	Neuman, B. C., and Ts'o, T. "Kerberos: An Authenication Service for Computer Networks". *IEEE Communications*, Vol. 32, No. 9, pp. 33-38. September 1994.
[NGIO99]	Next Generation I/O Forum, "NGIO 1.0 specification," July 1999.
[NIST94]	National Institute of Standards and Technology, FIPS 186, *Digital Signature Standard*, May 1994.
[NIST95]	National Institute of Standards and Technology, FIPS 180-1, S*ecure Hash Standard*, April 1995.
[NIST98]	National Institute of Standards and Technology, *Advance Encryption Standard Development Effort Webpage*, csrc.nist.gov/encryption/aes/aes_home.htm, 1998.
[NIST99]	National Institute of Standards and Technology, FIPS 46-3, *Data Encryption Standard*, Draft Standard, January 1999.
[NSIC96]	National Storage Industry Consortium, *Network Attached Storage Devices Project*, `www.nsic.org/nasd`, 1996.
[OceanLogic99]	Ocean Logic Pty Ltd., *DES core*, `www.users.bigpond.com/oceanlogic/des.htm`, January 1999.
[Organick72]	Organick, E., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
[OSF91]	Open Software Foundation, *DCE Application Development Guide*, Revision 1.0, Cambridge, MA, 1991.
[Ousterhout85]	Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J., "A Trace-Driven Analaysis of the UNIX 4.2 BSD File System," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, December 1985.

[Ousterhout88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., and Welch, B., “The Sprite Network Operating System,” *IEEE Computer*, February 1988, pp. 23-36.

[Pai99] Pai, V. S., Druschel, P., and Zwaenepoel, W., “IO-Lite: A Unified I/O Buffering and Caching System,” *Proceedings of the Third Operating System Design and Implementation Symposium*, February 1999.

[Palmchip99] Palmchip Corporation, *GreenLite IIp (HDD-1051-1.2) Data Sheet*, January 25, 1999.

[Patterson88] Patterson, et al., “A Case for Redundant Arrays of Inexpensive Disks (RAID),” *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, June 1988, pp. 109-116.

[Patterson95] Patterson, R. H., et al., “Informed Prefetching and Caching,” *Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.*

[Poole97] Poole, L. *Macworld Mac OS 8 Bible*, IDG Books Worldwide Inc. Foster City, CA, 1997

[Preneel95] Preneel, B. and van Oorschot, P. C., “MDx-MAC and building MACs from hash functions,” *Advances in Cryptology (CRYPTO 95), Lecture Notes in Computer Science Vol. 963*, Springer-Verlag, 1995, pp. 1-14.

[Preneel98] Preneel, B., Rijmen, V., and Boosselaers, A., “Principles and performance of cryptographic algorithms,” *Dr. Dobb’s Journal*, Vol. 23, No., 12, December 1998, pp. 126-131.

[Quantum99] Quantum Corporation, *ThinkBIGGER White Paper*, `www.quantum.com/src/whitepapers/think_bigger`, 1999.

[Rabin89] Rabin, M., and Tygar, J.D., “ITOSS: An Integrated Toolkit for Operating System Security,” *Proceedings of Third Internatonal Conference on the Foundations of Data Organization and Algorithms, Lecture Notes in Computer Science Vol. 367*, Springer-Verlag, 1989.

[Ramakrishnan98] Ramakrishnan, R., *Database Management Systems*, McGraw-Hill, 1998.

[Reed99] Reed, B., Chron, E., Long, D., and Burns, R., “Authenticating Network Attached Storage,” To appear: *Proceedings of Hot Interconnections ‘99*, 1999.

[Reiher93] Reiher, P., Page, T., Popek, G., Cook, J., and Croker, S., “Truffles - Secure Filesharing for Widespread File Sharing,” *Proceedings of the Privacy and Security Research Group Workshop on Network and Distributed System Security*, February 1993.

[Riedel96] Riedel, E., and Gibson, G., “Understanding Customer Dissatisfaction with Underutilized Distributed File Servers,” *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*. College Park, MD. September 1996.

[Riedel98a] Riedel, E., Gibson, G., and Faloutsos, C., “Active Storage for Large-Scale Data Mining and Multimedia,” *Proceedings of the 24th International Conference on Very Large Databases*, New York, NY, August 1998.

[Riedel98b] Riedel, E., van Ingen, Catharine, and Gray, J., “A Performance Study of Sequental I/O on Windows NT,” *Proceedings of the Second Usenix Windows NT Symposium*. Seattle, WA, August 1998.

[Rivest91] Rivest, R. “The MD4 Message Digest Algorithm,” *Advances in Cryptology (CRYPTO 90), Lecture Notes in Computer Science Vol. 537*, Springer-Verlag, 1991, pp. 303-311.

[Rivest92] Rivest, R., *The Message-Digest Algorithm*, Internet RFC 1321, April 1992.

[Rummler91] Rummler, C., and Wilkes, J., *Disk Shuffling,* Hewlett-Packard Laboratories Concurrent Systems Project, Technical Report HPL-CSP-91-30, 1991.

[Rutstein97] Rutstein, C. B., *National Computer Security Association Guide to Windows NT Security: A Practical Guide to Securing Windows NT Servers and Workstations*, McGraw-Hill, New Yrok, 1997.

[Ryutov98] Ryutov, T., and Neuman, B. C., *Access Control Framework for Distributed Applications*, Internet Draft, CAT Working Group, 1998.

[Sachs94] Sachs, M.W. et al.,“LAN and I/O Convergence: A Survey of the Issue,” *IEEE Computer*, December 1994.

[Sandberg85] Sandberg, R. et al., “Design and Implementation of the Sun Network Filesystem,” *Proceedings of the Summer 1985 USENIX Conference*, June 1985, pp. 119-130.

[Satyanarayanan89] Satyanarayanan, M., “Integrating Security in a Large Distributed System,” *ACM Transaction on Computer Systems*, Aug. 1989, Vol. 7, No. 3, pp. 247-280.

[Schlosser98] Schlosser, S., and Schmidt, B., Personal Communication, December 1998.

[Schlumberger99] Schlumberger Corporation, *Cyberflex Product Page*, `www.cyberflex.com`, January 1999.

[Schneier96] Schneier, B., *Applied Cryptography,* John Wiley & Sons, Inc., 1996.

[Schneier97] Schneier, B., and Whiting, D., “Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor,” *Proceedings of the Fourth International Workshop on Fast Software Encryption*, Springer-Verlag, 1997, pp. 242-259.

[Schneier99] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and Ferguson, N., *Performance Comparison of the AES Submissions*, Version 1.4a, `www.counterpane.com`, January 1999.

[Schmit97] Schmit, H., “Incremental Reconfiguration for Pipelined Applications,” *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 47-55.

[Schuba97] Schuba, C., Krsul, I., Kuhn, M., Spafford, E., Sundaram, A., and Zamboni, D., “Analysis of a Denial of Service Attack on TCP,” *Proceedings of the IEEE Symposium on Security and Privacy* , 1997, Oakland, CA

[Seagate98] Seagate Corporation, *Storage Networking: The Evolution of Information Management*, `www.seagate.com/corp/vpr/literature/papers/storage_net.shtml`, 1998.

[Seagate99a] Seagate Corporation, *Online Product Listing - Disk Drives*, `www.seagate.com/cda/disc/guide/`, 1999.

[Seagate99b] Seagate Corporation, *Jini: A Pathway for Intelligent Network Storage*, `www.seagate.com/corp/vpr/literature/papers/jini.shtml`, 1999.

[Sheldon96] Sheldon, T., *Netware 4.1: The Complete Reference, Second Edition*, Osborne McGraw-Hill, Berkeley, California, 1996.

[SICAN99] SICAN Microelectornics Corporation, `www.sican.com`, January 1999.

[Sidhu90] Sidhu, G. S., Andrews, R. F., and Oppenheimer, A. B., *Inside AppleTalk, Second Edition*, Addison-Wesley, New York, 1990.

[Siemens97] Siemens, *TriCore News Release: Siemen’s New 32-bit Embedded Chip Architecture Enables Next Level of Performance in Real-Time Electronics Design*, `www.tri-core.com`, September 1997.

[Smith98] Smith, S. and Weingart, S., “Building a High-Performance, Programmable Secure Coprocesor”. IBM Research Report RC 21102, February 1998.

[SNIA98] Storage Networking Industry Association, `www.snia.org`, 1998.

[Stamos84] Stamos, J. W., “Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory,” *ACM Transactions on Computer Systems*, Vol. 2, No. 2, pp. 155-180, 1984.

[Steen96]	Steen, W., and Bierer, D., *NetWare Security*, New Riders Publishing, Indianapolis, Indiana, 1996.
[STK94]	Storage Technology Corporation, "Iceberg 9200 Storage System: Introduction," STK Part Number 307406101, Storage Technology Corporation, 1994.
[Sun88a]	Sun Microsystems, Inc. *RPC: Remote Procedure Call Protocol Specification*, Internet RFC 1050, April 1988.
[Sun88b]	Sun Microsystems, Inc. *RPC: Remote Procedure Call Protocol Specification Version 2*, Internet RFC 1057, June 1988.
[Sun89]	Sun Microsystems, Inc., *NFS: Network File Systems Protocol Specification*, Internet RFC 1094, March 1989.
[Sun97]	Sun Microsystems, Inc. *ONC+ Developer's Guide*, `docs.sun.com/ab2/coll.45.4/ONCDG/@Ab2TocView/1363`, 1997.
[TPC98]	Transaction Performance Council, *TPC-C Executive Summaries*, `www.tpc.org`, May 1998.
[Tanenbaum86]	Tanenbaum, A. S., Mullender, S. J., and van Renesse, R., "Using Sparse Capabilities in a Distributed System," *Proceedings of the Sixth International Conference on Distributed Computing*, 1986, pp. 558-563.
[Thekkath97]	Thekkath, C., et al., "Frangipani: A Scalable Distributed File System," *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997, pp. 224-237.
[Transarc91]	Transarc Corporatiom, *AFS: Programmer's Rerference Manual*, October 1991.
[Transarc92]	Transarc Corporation, *AFS: System Administrator's Guide*, July 1992.
[Tygar99]	Tygar, J. D., Personal Communication.
[vanMeter96]	van Meter, R., Hotz, S., and Finn, G., "Derived Virtual Devices: A Secure Distributed File System Mechanism," *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, September 1996.
[vanPelt99]	van Pelt, P., Marketing Manager, Pijinenburg Custom Chips B.V., Personal Communications, January 1999.
[Varma95]	Varma, A., and Jacobson, Q., "Destage Algorithms for Disk Arrays with Non-Volatile Caches," *22nd ISCA*, 1995.

[vonEicken95] von Eicken, T., Basu, A., Buch, V. and Vogels, W. “U-Net: A User-Level Network Interface for Parallel and Distributed Computing,” *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[VLSI99] VLSI Technology,V*LSI 115 Datasheet Version 2.0*, January 1999.

[Wahbe93] Wahbe, R., Lucoo, S., Anderson, T.E., and Graham, S.L., “Efficient software-based fault isolation,” *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993, pp. 203-216.

[Wania99] Wania, X., President & CEO Xentec Inc., `www.xentec-inc.com`, Personal Communication, January 1999.

[Webster85] Webster, A., and Tavares, S., “On the design of S-boxes,” *Advances in Cryptology (EUROCRYPT ‘95), Lecture Notes in Computer Science Vol. 750*, pp. 575-586, 1995.

[Weingart99] Weingart, S. H., Personal Communication, January 1999.

[Weingart87] Weingart, S. H., “Physical security for the μAbyss system”. *Proceedings of the IEEE Computer Science Conference on Security and Privacy*, pp. 52-58, 1987.

[Whites87] White, S. R., and Comerford, L., “ABYSS: A trusted architecture for software protection”. *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pp. 38-51, 1987.

[White91] White, S. R, Weingart, S. H., Arnold, W. C., and Palmer, E. R., *Introduction to the Citadel arcihtecture: Security in physically exposed environments*, Distributed security system group, IBM Thomas J. Watson Research Center, Technical Report RC16672 March 1991.

[Wilkes92] Wilkes, J. *Hamlyn - An Interface for Sender-based Communications*, Hewlett-Packard Laboratories Technical Report HPL-OSR-92-13, November 1992.

[Wilkes95] Wilkes, J., et al., “The HP AutoRAID Hierarchical Storage System,” *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[Wolfe99] Wolfe, A., “Sun’s Jini completes for post-PC paydirt,” *EE Times Online*, `www.eet.com/story/OEG19990122S0011`, January 26, 1999.

[Yee95] Yee, B., and Tygar, J. D., “Secure Coprocessors in Electronic Commerce Applications,” *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.

[Yee94] Yee, B., *Using Secure Coprocessors*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-94-149, 1994.

[Zimmerman95] Zimmerman, P., *The Official PGP User's Guide,* MIT Press, 1995.