USENIX Association

# Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies

San Francisco, CA, USA
March 31–April 2, 2003

# Block-Level Security for Network-Attached Disks

Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli,
Dave Andersen, Mike Burrows, Timothy Mann, Chandramohan A. Thekkath
*HP Systems Research Center**
*Palo Alto, CA*

## Abstract

We propose a practical and efficient method for adding security to network-attached disks (NADs). In contrast to previous work, our design requires no changes to the data layout on disk, minimal changes to existing NADs, and only small changes to the standard protocol for accessing remote block-based devices. Thus, existing NAD file systems and storage-management software could incorporate our scheme very easily. Our design enforces security using the well-known idea of self-describing capabilities, with two novel features that limit the need for memory on secure NADs: a scheme to manage revocations based on *capability groups*, and a replay-detection method using Bloom filters.

We have implemented a prototype NAD file system, called *Snapdragon*, that incorporates our ideas. We evaluated Snapdragon's performance and scalability. The overhead of access control is small: latency for reads and writes increases by less than 0.5 ms (5%), while bandwidth decreases by up to 16%. The aggregate throughput scales linearly with the number of NADs (up to 7 in our experiments).

## 1 Introduction

Network-attached disks (NADs) are storage devices that accept block read/write requests over the network. They can be used to build file systems that provide better performance than traditional distributed file systems such as NFS [22]. In traditional systems, disks are attached directly to a file server, which provides file access to clients across a network. Because all data must pass through the server, it quickly becomes a bottleneck as the system scales, limiting the achievable bandwidth.

NAD file systems, in contrast, allow clients to bypass the file server and go straight to disk to read and write file data. Although clients must still talk to the file server for metadata operations (*e.g.*, file lookup and deletion), the file server's bandwidth no longer constrains the file system's bandwidth. Such *asymmetric shared file systems* [18] excel at workloads with high bandwidth and relatively few metadata operations. Commercial examples of such file systems include Tivoli's SANergy [25] and SGI's CXFS [11].

Network-attached disks commercially available today do not provide any support for security: they honor any request received. Thus, securing a NAD file system today requires the NAD network and all attached machines and disks to be physically secured. In practice, this forces the use of a separate LAN for storage (usually called a Storage Area Network, or SAN) and prevents clients located outside a machine room or under end-user control from directly accessing the NADs. Unfortunately, this means that NAD file systems cannot deliver high bandwidth to desktop machines, preventing many useful applications (*e.g.*, supplying training videos to PCs and desktop data mining) from taking advantage of NAD technology.

We believe that any practical approach to this problem should minimize the changes required in order for commercial NAD file systems to use it. Schemes that require major changes to commercial file systems, or the creation of a commercial-quality file system from scratch—both very expensive propositions—are unlikely to be adopted. Also of concern are the changes needed to storage-management software, such as monitoring, mirroring and backup tools.

Unfortunately, existing approaches that add security to NADs require large changes: NASD [7, 8] replaces the existing NAD block model with a variable-length data-object model and moves most of the filesystem functionality onto the disk. SUNDR [13] indexes blocks by their cryptographic hash instead of their (logical) position on the disk and garbage collects blocks whose *required-by* lists become empty. SNAD [5, 15] disks use special file

*This work was done at the HP Systems Research Center in Palo Alto. The authors with differing current affiliations are: Andersen, MIT; Burrows and Thekkath, Microsoft Research; Mann, VMware.

and key objects to keep track of which users can access each block—in essence, access control lists must be stored in a particular format so as to be understandable to the disk. SNAD also stores a small client signature (36-100 bytes) for each block, requiring it to either offer clients a non-power-of-two raw block size or suffer a substantial performance penalty.

We propose in this paper adding simple block-level security to network-attached disks. Our proposal maintains the existing NAD sequentially-numbered raw-block view of storage, allowing existing NAD clients to continue using their existing data layout and management strategies (*e.g.*, block-based backup); the only changes required are to create and pass along authorization information from the server through the clients to the disks. Using a raw-block view allows for maximum flexibility—higher-level primitives often limit what can be done. For example, it seems difficult to implement a file system that can take atomic snapshots on top of SNAD's built-in primitives.

By having NADs verify that a request comes unaltered from a client authorized to read or write that block, and by encrypting network traffic, we can provide a reasonable level of security even in an environment where end users control client machines and the network is vulnerable to attacks such as wiretapping and spoofing. Under our system, requests are authorized by an appropriate accompanying capability. Similarly to the NASD security approach [10], our central server, which we call a *metadata server*, issues and manages capabilities, setting policy, while the disks do only simple access checks.

Because the naive approach of one capability per block has too much overhead, each of our capabilities specifies access to one or more ranges of blocks (*extents*). Since we do not want our disks to have to understand which blocks belong to which files, we use *self-describing capabilities*: the accessible blocks and access mode can be determined from a capability without reference to any other data structures.

To allow revocation of capabilities, we require disks to remember which capabilities have been invalidated; this validity data must be held in RAM for speed reasons. An important contribution of our work is our revocation method based on *capability groups*, which dramatically compresses the validity data without sacrificing performance. Another contribution is our non-connection-based method of handling replay attacks, which allows for an unlimited number of clients while using only a small amount of memory.

Because of these techniques, our disk protocol's requirements on the NADs are very low: standard cryptographic functionality plus a small amount of RAM for capability management and replay detection (on the order of 128 KB). We believe this requirement is so small that the scheme could be deployed in existing NADs by simply changing their firmware, without modifying or adding hardware. By comparison, existing approaches to adding security to NADs require much deeper changes to the disks, and they would cost significantly more to implement.

The remainder of the paper is organized as follows: Section 2 describes our basic scheme to achieve security. Section 3 describes the implementation of a prototype NAD file system we built to demonstrate our scheme. Section 4 explains some other important aspects of our design. Section 5 discusses the prototype's performance, including the cost of security and the prototype's scalability. Section 6 covers some limitations of our design. Section 7 covers related work. And, finally, Section 8 concludes the paper.

# 2 Block-based security with modest RAM

We assume that the server and disks can be trusted—they are responsible for setting and enforcing security policies under our approach. However, we assume clients can be compromised and that the network is not secure, either from eavesdropping or spoofing. Under these conditions, our security ensures that attackers can access a user's data only by compromising a client machine logged into by that user. Should encryption be turned off for performance reasons, some privacy will be lost, as a wiretapper can see data read or written. The level of security we offer is similar to that of an NFS system that uses Kerberos for authentication, cryptographic checksums for integrity, and optional encryption for privacy.

Unforgeable, self-describing capabilities are the chief mechanism we employ for adding security to a NAD file system. We use the well-known idea of capabilities composed of two parts: a self-describing certificate and an associated secret. The secret is generated via a message authentication code (MAC) from the certificate and a hidden key known only to the server and the relevant disk [10, 9, 16, 19]. This basic capability approach, reviewed in Section 2.1 below, is augmented by two new techniques which permit RAM requirements on the system's disks to be very modest: a revocation scheme based on *capability groups*, described in Section 2.2; and a defense against replay attacks using Bloom filters, described in Section 2.4.

| group ID | capability ID | disk ID | extents | mode |
|---|---|---|---|---|

Figure 1: **Contents of a capability.** The group ID and capability ID are used in our new revocation scheme. The disk ID, extents, and mode describe the access granted by the capability.

## 2.1 The basic capability scheme

Our protocol for using capabilities is similar to that of NASD [10, 9] and SCARED [19], except that our capabilities describe access in terms of blocks rather than objects. Intuitively, a capability is a self-descriptive certificate that grants a specified type of access to parts of a disk (see Figure 1) when used with an associated secret. Our capabilities contain a group ID and a capability ID, which are used for revocations as explained in Section 2.2; a disk ID, which specifies the disk to which this capability applies; a list of *extents*, the ranges of physical blocks for which access is being granted; and an access mode (read, write, or both).

The secret is used to prevent forgery of illegal capabilities or of illegal requests using legal capabilities, as we now explain. The secret is generated using a *keyed-hash message authentication code*, or MAC [1]. A MAC function $h(\mathsf{data}, \mathsf{key})$ returns a string mac of fixed length with the following *unforgeability property*: without knowing the value of key, it is infeasible to find *any* new pairs of mac and data such that $\mathsf{mac} = h(\mathsf{data}, \mathsf{key})$. MAC functions can be computed quite efficiently in practice, unlike public-key signatures.

Every capability $c$ is associated with a secret $h(c, k)$, where $k$ is a secret key shared by the metadata server and the disk whose ID is specified in $c$. (There is a different key for each disk.) The use of this secret is best explained by an example. Figure 2 shows a client opening a file for the first time, and then reading or writing some data. To open the file, the client contacts the metadata server associated with the file. If the file's metadata is not cached at the server, the server must retrieve it from the relevant disk, shown by dashed lines in the figure (the metadata server accesses the disk in the same way that the client does, which we explain below).

The server checks if the client is permitted to access the file, and if so it gives the client the following: (1) the list of physical blocks comprising the file (its *blockmap*), (2) a capability $c$ for the file's blocks[1] with the requested access mode (read, write, or both), and (3) the secret

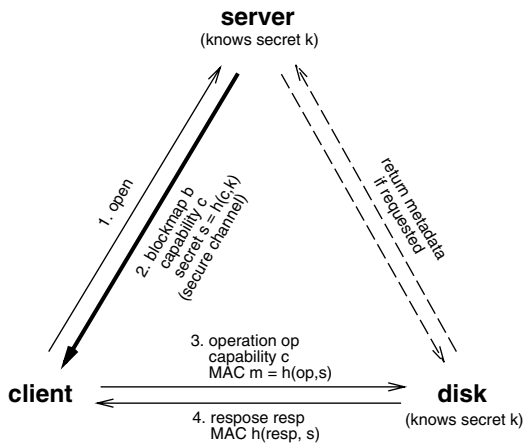


Figure 2: **Opening and accessing a file.** When a client wishes to access a file, it talks to the metadata server to get a capability $c$ and its associated secret $s = h(c, k)$. The client can then access the file directly from disk. The disk verifies that the access is authentic by checking that the client has correctly computed the "double MAC" $m = h(op, s) = h(op, h(c, k))$.

$s = h(c, k)$ associated with $c$. The server's reply (in particular $s$) must be sent over a secure channel (shown by a darker line in the figure) to prevent eavesdroppers from learning the secret needed to use the capability. A secure channel can be obtained by encrypting under a block cipher using a session key established by an authentication protocol such as Kerberos.

Next, in order to read or write data from or to the file, the client issues block requests to the disk using the capability that it obtained. More precisely, the client sends to the disk an operation $op$ that consists of (1) the type of access (read or write); (2) the range of physical blocks to be accessed; and (3) in case of a write operation, the data to be written. Together with $op$, the client also sends the capability $c$ provided by the server and a MAC $m = h(op, s)$, where $s$ is the secret associated with $c$. Because $m = h(op, s) = h(op, h(c, k))$, we call this trick the "double MAC". (The double MAC is not new; the earliest references we know of are Gobioff *et al.* [10] and Mittra *et al.* [16].) The disk can verify that the MAC is correct since it receives both $c$ and $op$, and it has the secret key $k$. Note that the double MAC serves a double purpose: (1) it is a proof that the client knows $s$ and thus has been authorized to use the capability $c$ to issue the operation $op$, and (2) it prevents $op$ from being tampered with, because if an attacker changes $op$ it would not know how to compute the required new MAC.[2]

[1] For simplicity, this example assumes that the file's blocks can be described using only one capability; in practice, highly fragmented files may require multiple capabilities because capabilities have a fixed size so that they can fit in a packet.

[2] The reader might be wondering whether the application of a MAC to its own output has somehow compromised its cryptographic properties. This is not the case, and the intuitive argument is as follows: Sup-

Once the disk has checked and executed the requested operation $op$, it sends back a response $resp$ together with $h(resp, s)$. Here $resp$ contains data (if the request was a read) or simply an acknowledgment (if the request was a write). The client verifies that $h(resp, s)$ was computed correctly, which prevents responses from being forged.

For simplicity, we presented this example without encryption for privacy. One simple way of adding encryption involves the server also giving the client a session key $e$ and a token, which is $e$ encrypted under $k$; the client and disk encrypt their messages using $e$, prepending the token in the clear so the disk can figure out which session key to use.

## 2.2 Revoking capabilities

A revocation is required whenever a client should no longer have the access granted by a previously issued capability—due, for example, to a change in file permissions, or a file truncation or deletion. We seek a revocation scheme that is memory efficient, so that it can ideally be implemented in existing network-attached disks by simply changing their firmware (rather than, say, adding more hardware to them). The difficulty is that such disks have little memory and most of it is used to cache data (a typical cache size is 4 MB). It is thus important that the adopted scheme not take much memory: tens of kilobytes would be excellent, while megabytes would probably be too much.

Earlier schemes described in the literature, such as NASD [10] and SCARED [19], use object version numbers for revocations. In these systems, capabilities confer access only to a particular object version, so incrementing an object's version number suffices to revoke all old capabilities for it. Although this makes sense for variable-length objects, whose headers must be read first to find out where the desired data actually resides on disk, it is problematic for blocks: changing the permissions of a file would require updating a potentially large number of version numbers. For example, a 512 MB file could require updating 1 million version numbers, which would span 8 thousand blocks assuming 32 bits per version number.

Thus, we need a scheme that allows direct revocations of capabilities, without having to access the blocks to which the capability refers. A simple and economical method is to assign capability ID's to each capability so that the disk can keep track of valid capabilities through a bit vector. By doing so, it is possible to keep track of 524,288 capabilities with 64 KB, a modest amount of memory.

As a further optimization, we can assign the same ID to different capabilities, thereby reducing the number of possible revocations that need to be kept track of, but at the cost of not being able to independently revoke capabilities that share an ID. It makes sense to group together all the capabilities describing the same kind of access to different parts of one file, because they are almost always revoked together (the exception being partial truncation). One can further group together capabilities for the same file with different access modes; this makes file creation and deletion cheaper at the cost of making permission changes (*e.g.*, `chmod`), which are rare, slightly more expensive.

This straightforward approach has a problem though: once an ID is revoked, its bit in the revocation vector must be kept set *forever*, lest some attacker hold the revoked capability and much later illegally use it again if the bit were cleared. Therefore, no matter how large the number of ID's, the system will sooner or later run out of them.

One way to solve this difficulty is to change $k$ (the secret key shared by the disk and metadata server) whenever ID's run out, causing all existing capabilities to become stale. The disk's bit vector can now be cleared without fear of old invalidated capabilities springing back to life and posing a security threat.

This scheme has the unfortunately problem that when $k$ is changed all capabilities become stale at once and will be rejected by the disk, so all clients need to go back to the metadata server to get fresh capabilities. Therefore, the metadata server may get overloaded with a shower of requests in a short period. We call this the *burstiness problem*.

We solve the burstiness problem by using *capability groups*. The basic idea is to place capabilities into groups, and to invalidate groups when the system needs to recycle ID's. Intuitively, this avoids the burstiness problem because only a *small fraction* of capabilities is revoked when system runs out of ID's. More precisely, each capability has a capability ID and a group ID. The disk keeps a list of valid group ID's, and for each valid group, a bitmap with the revocation state of ID's. To know if a capability is still valid, the disk checks if its group ID is valid and, within that group, whether the capability ID is not revoked. To recycle the capability ID's

pose that it was feasible for an adversary who does not know the value of $k$ to produce values $m$, $op$, and $c$ such that $m = h(op, h(c, k))$. Then by the unforgeability property of $h$ mentioned above, the adversary must know the value of $h(c, k) = s$. Thus, the adversary can produce $c$ and $s$ such that $s = h(c, k)$. Therefore, by applying the unforgeability property again, the adversary must know the secret key $k$, a contradiction.

| Group ID | | Revoked capability ID's |
|---|---|---|
| Index | Counter | (bitmap) |
| 0 | 10 | 100010001000 . . . |
| 1 | 5 | 001111011010 . . . |
| 2 | 14 | 111011111000 . . . |
| ⋮ | ⋮ | ⋮ |
| 63 | 3 | 000011010111 . . . |

Figure 3: **Keeping track of revocations.** The table used by the disk controller to keep track of revoked capabilities.

in a group, the group ID is removed from the list of valid groups and it is replaced with a new group ID.[3] Then, all capability ID's of the new group are marked valid.

In our particular implementation, we divide a group ID into two parts: a 6-bit index and a 64-bit counter. The index part is used to index a 64-entry table, each entry of which contains a counter and 8,128 bits of revocation data (see Figure 3). The table requires $64 \times (8128 + 64)$ bits or 64 KB of RAM and supports up to $64 \times 8128 = 520,192$ simultaneous capabilities. A capability is checked by looking up the entry corresponding to the index part of its group ID, and verifying that the counter matches the one in the capability's group ID. If so, the bit corresponding to the capability ID is tested. Revocation of a capability is done similarly. Group invalidation is done by clearing the group's bitmap and incrementing its counter, effectively replacing its group ID with a fresh new one. All these operations are very quick (small constant time) and space efficiency is excellent: each capability takes on average less than 1.01 bits.

Note that capability groups alone do not reduce the total work on the metadata server over time, but the work gets spread over a longer period, avoiding the burstiness problem. We have done simulations with real trace data that confirm our predictions. Our simulations show that the peak load on the metadata server is reduced significantly with this technique (see Section 4.1 for more details).

## 2.3 Network partitions

When a network partition separates the metadata server from a disk, the server is unable to revoke capabilities for that disk, resulting in the access permissions of files on that disk effectively being frozen; in some systems, this could be considered a security breach. To avoid this problem, we can require the metadata server to periodically refresh the table of groups and capabilities of each disk. If a disk does not receive a refresh message within a certain period of time, it disallows all accesses until it receives the expected server refresh.

Of course, such a scheme can be disabled if the system administrator believes that the overhead of the refresh messages is too high for the protection it provides.

[3] Note that the group ID's *cannot* be recycled, which means that in theory the system will eventually run out of space. But by using relatively few bits for the group ID's—say 64 bits—it will take longer than the life of the system for that to happen.

## 2.4 Preventing replay attacks

While it is infeasible for an adversary to forge new requests, it is trivial to replay requests that have already been sent to a disk. Hence, a NAD file system that operates using an unsecured network must have robust defenses against replay attacks. (Note that replay attack prevention is harder than duplicate detection [2, 12], which assumes all parties are honest.) Fortunately, it is possible to achieve this at low cost in memory and computation, without requiring per-client information. The method, which we believe is novel in the context of replay attacks, employs a data structure called a Bloom filter [4] to remember recent requests.

Bloom filters are a highly efficient way of performing approximate set-membership queries; given a membership query, they answer either "probably an element" or "definitely not an element". A Bloom filter consists of an array of $K$ bits, denoted $b_1, b_2, \ldots, b_K$, together with $n \geq 1$ hash functions, $f_1, \ldots, f_n$. The hash functions are chosen randomly from a family of independent hash functions at filter construction time; each maps requests to integers in $\{1, 2, \ldots, K\}$. The filter is defined to be *empty* when all bits are 0. A request $r$ is added to the filter by setting the bits with indices $f_1(r), f_2(r), \ldots, f_n(r)$—*i.e.*, we set $b_{f_i(r)} = 1$, for all $i$. To answer the question "is request $r$ in the filter?", we reply "probably" if $b_{f_i(r)} = 1$, for all $i$, and "definitely not" otherwise.

A disk can detect replays by keeping a list of seen requests in a Bloom filter. When a new request arrives, the disk checks to see if it is already in the filter. If the filter reply is "definitely not", the disk can safely proceed to process the request after adding it to the filter as it cannot be a replay. Otherwise, it is likely that the request has already been issued in the past, so the disk sends a replay rejection message. The client continues to retransmit a request until it receives either an acceptance or rejection message for that request. If it gets a reply rejection message, it changes the request's nonce (so that it hashes differently) and continues retransmitting. For the nonce, we

use a small sequence number, which serves no other purpose. Note that in a system with message losses, a client may sometimes end up executing its own request multiple times consecutively, but this is not a problem when requests are idempotent.

Of course, after enough requests have been added, the filter will begin to have a non-negligible false-positive rate. We consider a filter in need of replacement when more than a fixed proportion of its bits are set. We implement filter replacement by maintaining several filters at the disk together with a monotonically-increasing *epoch number*, which is periodically checkpointed to disk. Each filter is associated with a recent epoch. When the filter corresponding to the current epoch needs replacement, the disk increments its epoch number, deletes its oldest filter, and starts a new filter to handle the new epoch. On reboot, the epoch number is incremented by the number of filters; this prevents replaying messages sent while the disk was down.

A client sends what it believes to be a disk's latest epoch number—each disk message includes the current number—with every message to the disk. If the epoch number in a client request is too old (*i.e.*, more out of date than the number of filters being maintained), the request is rejected. Otherwise, it is checked against the appropriate Bloom filter. In this way, the switch to a new filter can be made transparent to active clients of the disk. (Clients idle sufficiently long will have their first request rejected due to its out-of-date epoch number.)

It is worth pointing out the following optimization: instead of applying the hash functions to the whole request $r$, which can be quite large (*e.g.*, it includes the data in a write operation), it suffices to apply them to just the request MAC $m = h(op, s)$ (described in Section 2.1), which is only a few bytes long. Note that the request epoch number and nonce are included in the operation $op$, which is guarded by the MAC, preventing an attacker from altering them.

Another optimization involves not storing read requests in the Bloom filter, allowing for even smaller filters. Note that read requests need only be checked if encryption is turned off. And even in that case, it may not be necessary to check *recent* read requests, because the attacker could have snooped on the reply of the original read. Thus, only very old read requests need to be filtered out, and this can be accomplished by simply verifying that the request's epoch number is valid; there is no need to use the Bloom filter at all. (Epoch numbers should be periodically advanced with this optimization.) Our performance numbers do not include this optimization.

Our method to prevent replay attacks with Bloom filters is simple, robust, and frugal. In contrast, existing methods such as [3], which keep per-client state, have two drawbacks: (1) they can support only a limited number of clients when constrained to use similarly small amounts of memory, and (2) they require the extra complexity of authenticating clients to the disk to guard against a rogue client claiming too large a share of the client-state table.

Our approach is also different from NASD, which relies on a real-time disk clock and expiration times instead of an epoch number that can be bumped at any time. Unfortunately the NASD scheme limits the maximum rate of requests that NASD can handle to the maximum size of its recent-request list (stored using a less space-efficient array) divided by its expiration time [9]. This could be a problem if many requests hit the disk cache.

## 2.5 Consistency attacks

If leases (*i.e.*, locks with timeouts) are used to cache filesystem data at the clients, an attacker that wishes to create consistency problems can proceed as follows: A client gets a lock for a file and issues a write request. The attacker then launches a denial-of-service attack to simultaneously capture and obliterate the write request (and subsequent retries) so that it never reaches the disk. After the lock has expired, the attacker sends the captured write request to the disk, which executes the write without the lock held, potentially causing consistency problems.

To guard against this type of attack, the system could invalidate requests that are outstanding when the lock expires. To do that, the metadata server could revoke all capabilities issued to the client that holds the expired lock; the metadata server then waits until the disk has acknowledged the revocations before it breaks the lock.

## 2.6 Data structures and disk functionality

Our addition of security to a NAD file system requires several data structures, which are listed in Figure 4. At the metadata server, we maintain a hash table of all valid capabilities for use in performing revocations: whenever the access to a file changes, we need to find all capabilities associated with that file and revoke them. The server also maintains copies of each disk's valid group list plus the number of valid and revoked capabilities in each group so that it can quickly choose which group to invalidate next.

**At the metadata server:**

- hash table of all currently valid capabilities, indexed by inode number
- for each disk, a list of valid groups, with the number of valid and revoked capabilities in each

**At a client:**

- a cache of capabilities issued to this client that are not known to have been revoked

**At a disk:**

- one counter and bitmap per valid group (64 KB)
- Bloom filters of recent requests (64 KB)

Figure 4: **Additional data structures for security.**

| new functionality | equivalent lines of C |
|---|---|
| cryptography | 340 |
| capability groups and revocations | 60 |
| miscellaneous (refresh timer, RPC handlers, logging) | 610 |

Figure 5: **Additional disk functionality.** The left column describes the purpose of the additional functionality that would be required on a secure disk; the right column gives the number of lines of C devoted to that functionality in our software implementation.

Clients cache issued capabilities to cut down on metadata-server traffic. No invalidation protocol is needed because if a client uses a cached capability that is no longer valid, the disk will reject it, leading the client to request a new one from the metadata server. The data structures at a disk have already been discussed in the sections on capability management (Section 2.2) and replay attacks (Section 2.4).

Figure 5 lists the modest extra functionality required to add our block-based security to a NAD. Combined with the fact that the additional data structures could use as little as 128 KB of RAM (see Figure 4), this suggests that our approach requires minimal changes to disks.

# 3 Implementation

We have implemented a prototype NAD file system, called *Snapdragon*, that uses our security approach. To do so, we modified Linux's existing kernel-based implementation of NFS version 2. (We used version 2 as a base because version 3 is not available as a loadable module, hindering debugging.) NFS and its utilities comprise about 45,000 lines of C. To this we added: (1) new filesystem code comprising 7,500 lines (4,000 at the server and 3,000 at the client); (2) new disk functionality comprising about 1,000 lines (see Figure 5); and (3) a security library of about 14,000 lines, the vast majority of which was imported from openssl.

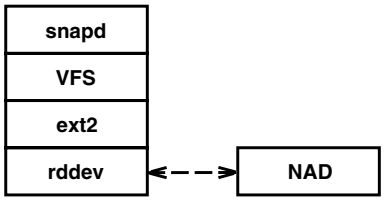


Figure 6: **Snapdragon server and NAD.** When the Snapdragon server (snapd) receives a client request, it passes it to Linux's VFS, which invokes the underlying filesystem (ext2 in our case), which in turn issues block requests to the device driver rddev. The latter translates these requests to NAD requests containing "allow-all" capabilities.

## 3.1 Overview

Snapdragon clients run two kernel modules: a standard NFS lock daemon and a module that contains the core filesystem functionality, including requesting and caching capabilities and (partial) blockmaps. The second module exports through Linux's Virtual File System (VFS) interface the new filesystem type "snapfs".

The Snapdragon metadata server consists of a filesystem kernel module (snapd), a device driver (rddev) and a lock daemon (lockd). Lockd is identical to NFS's. Snapd and rddev are shown in Figure 6. Snapd translates client requests into the filesystem-independent operations. In Linux such operations are handled by the VFS layer, which invokes filesystem-specific code (in our case ext2) that implements the operation by issuing low-level block requests to the device driver, rddev in this case. Rddev translates these block requests to messages to the disk controller (NAD) using the same protocol as the clients use, but using "allow-all" capabilities. This architecture allows Snapdragon to be independent of the underlying file system and allows the data layout on remote disks to be exactly the same as if the disks were local. This has nice implications for deployment as we explain in Section 4.3.

The Snapdragon disk controller is implemented as a PC connected to the network. The PC runs a small multithreaded user-level program that listens for, checks, and executes block requests.

## 3.2 Changes to the NFS protocol

The Snapdragon metadata server implements a superset of the NFS protocol. Snapdragon clients do not issue the NFSREAD and NFSWRITE RPCs to the metadata server, because reading and writing are handled locally at the client by issuing block read and write requests directly to the relevant NAD based on cached blockmaps and capabilities. Unmodified legacy NFS clients can continue to talk to a Snapdragon metadata server using standard NFS RPCs. Metadata consistency is ensured because all metadata commands are still handled by the server.

Three commands were added to the NFS protocol: OPEN, CLOSE, and GETCAPS. When issuing a GETCAPS call, the client passes a file handle, an access mode, and a range of logical blocks in the file, presumably because it would like to read or write these blocks in the future. The metadata server returns a blockmap specifying the corresponding physical blocks and capability(s) giving the client the requested access to the requested blocks. To keep messages within a reasonable size, if the range of requested blocks is very large or fragmented, the server returns a blockmap and set of capabilities which cover a range as large as possible, and the client must send another request for the remaining blocks.

Because GETCAPS returns a blockmap as well as capabilities, we can invalidate a file's blockmap by revoking the file's capabilities: any attempt by a client to use the old blockmap will result in an revoked-capability error from the disk, forcing the client to do a GETCAPS before it can proceed, giving it the new blockmap.

The OPEN and CLOSE commands are necessary only for caching file contents at the client, not for security. In our system, every open file is in either *exclusive* or *non-exclusive* mode. A file is in exclusive mode if either precisely one client has it open (reading or writing) or no client has it open for writing. When a client has a file open that is in exclusive mode, it knows that the file cannot undergo changes it is not aware of, and therefore it can cache the file's contents. In other situations, clients must use short timeouts on their cache in order to achieve acceptable levels of consistency. (We provide the same consistency as NFS.[4]) Because the server is notified whenever a client opens or closes a file, it knows when the exclusivity of a file changes, and can notify the clients that have that file open by using callbacks.

[4] A higher level of consistency could be implemented using the same basic technique as Spritely NFS [24]: channeling reads and writes for non-exclusive–mode files through the metadata server.

We take advantage of the need for an OPEN call, by piggybacking (on the reply to the client) a blockmap and capabilities for the opened file covering as many blocks as possible. This dramatically reduces the need for GETCAPS calls.

One might think that the separation of metadata and data in a NAD file system would require appending to a file be given special treatment. It turns out that append operations can be subsumed under standard write operations, by using Unix's *bmap* function with the *allocate* flag set. This function maps a logical block of a file to a physical block. If the block is not yet mapped and the allocate flag is set, the block is allocated according to the specifics of the underlying file system. Thus, to append to a file, a client issues a standard GETCAPS call for write access to logical blocks beyond the current end of the file. The metadata server can then simply call `bmap` to simultaneously allocate and get the newly appended block.

## 3.3 Capabilities

The design of our capability format was guided by studying the properties of the AdvFS file systems at our 30-person research laboratory. Recall that a capability includes a fixed-size list of extents, which are contiguous ranges of physical blocks. Files occupy one or more extents; for example, `/foo/bar` might occupy blocks [243-256], [9323-9992], and [20-50]. In our file systems, a single extent covers, on average, 150 KB. Moreover, it turns out that 90% of files require four or fewer extents, and that 95% require 13 or fewer extents. Hence, we decided that any single capability would have space allocated for four extents. Files with more than four extents can be accessed with multiple capabilities.

## 3.4 Bloom filter parameters

We use two 32KB bloom filters (262,144 bits each). We determined the other parameters by optimizing, using statistical simulation, for the maximum number of requests on average that can be supported per epoch subject to a maximum false-positive rate, measured over the last 1,000 requests, of 0.1%. The resulting parameters—$n = 9$ hash functions and about 47% of bits used in a full filter—yield epochs lasting 18,640 requests on average, or 30 minutes under the request rate of the trace used in Section 4.1.

### 3.5 Cryptographic details

Wherever a MAC is required, we use HMAC with the Secure Hash Algorithm SHA-1 [17]. This function returns 20-byte hashes, can be computed extremely fast, and possesses no known collisions. The client/server secure channel (see Figure 2) is achieved using the Blowfish block cipher algorithm [23] with a 16-byte key. When privacy is desired, we use DES encryption on messages to and from the disk.

Key management is rudimentary in the current prototype: all keys are read from configuration files and remain fixed indefinitely. Naturally, in a mature system, one could use a more elaborate scheme like the one described by Gobioff *et al.* [10].

### 3.6 Packet security overhead

Capabilities occupy 72 bytes using generous 64-bit values for block numbers. The replay epoch number has 64 bits, the nonce plus random padding for encryption use 128 bits, and the MAC has 160 bits. Thus, the total security data in a disk request is 116 bytes compared to up to 8192 bytes of payload.

## 4 Discussion

### 4.1 Practical benefit of capability groups

Is the capability group method beneficial in practice? In particular, do capability groups reduce the burstiness of capability allocations, thus reducing the chance of the metadata server being overloaded? To answer this question, we simulated the behavior of secure NADs using the trace of a 500 GB file system used by about 20 researchers over 10 days [21]. The results are shown in Figure 7, which is a histogram of how many capability allocations were performed by the metadata server in each 1-second period. The amount of memory for capability storage was fixed at 64 KB. The black bars show results for the straightforward method of Section 2.2, which uses a single bitmap to store all capabilities; this is precisely equivalent to the capability group method, with the number of groups $g = 1$. The white bars are for a configuration using the same amount of memory, but divided into $g = 64$ groups.

Note that for this particular workload, the metadata server could hardly be described as "overloaded": the peak load of around 500 capabilities/second can easily be handled since it takes less than 2 ms to issue a capability. Nevertheless, these results demonstrate that the capability group approach substantially reduces burstiness. If the stress on the server were greater (*e.g.*, if it served more disks), the capability group approach would improve performance by reducing periods of overload.

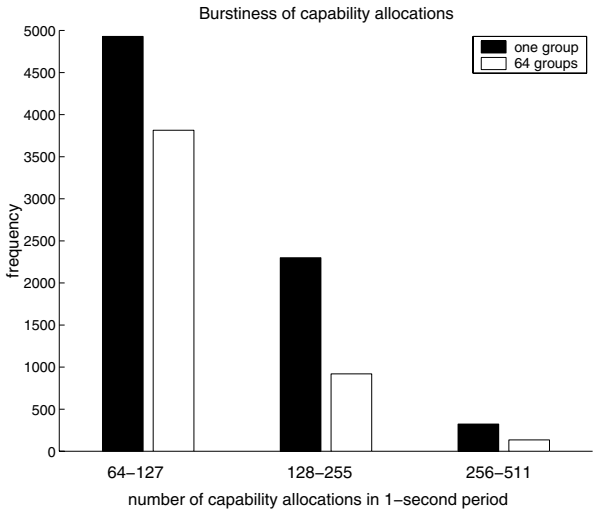


Figure 7: **Using 64 groups instead of 1 substantially reduces the burstiness of capability allocations.**

### 4.2 Choice of underlying file system

Our approach to security can be achieved by incrementally modifying most types of file systems. The major exceptions are file systems that store data from multiple files in the same block, such as ReiserFS [20] and FFS [14], which store the tails of multiple files in a single block. These file systems cannot be used because they require clients be given access to only part of a block, which our block-based capabilities cannot handle.

### 4.3 Deployment

Our approach is easy to deploy incrementally in existing legacy NFS environments. So long as the same underlying file system (and hence, disk layout) is kept, we could take existing disks with data and use them (without reformatting) as NADs by attaching them to the network via a controller that checks for capabilities and replay attacks. (Of course, this controller would have to be specially manufactured for use with our protocol.) By having our metadata server also export the Snapdragon file system via plain NFS, we can support legacy NFS clients, albeit

with poorer performance. This allows an NFS system to be converted one client at a time.

### 4.4 Optional improvements to capability groups

There are some improvements that could be applied to our basic technique of capability groups, which we now describe for completeness. However, for practical purposes we found that Snapdragon performed quite well even without these optimizations—in fact, our performance numbers in Section 5 do not include them.

Note that when a group is invalidated, there will be some *unintended revocations*, that is, valid capabilities will be revoked even though the permissions of their files have never changed. This of course does not break the correctness of the protocol: the client with an invalid capability can simply request a new capability from the metadata server. However, performance is affected because this procedure costs two extra network round trips: one when the client, unaware, attempts to use the invalid capability and gets rejected, and another round trip when the client requests the new capability.

It is desirable to minimize the number of such unintended revocations. An obvious strategy is to choose for invalidation a group with many revoked capabilities and few valid ones. In addition, one can exploit the fact that different capabilities have different (probabilistic) lifetimes. For example, a read-only capability for a shared library is unlikely ever to be revoked, whereas a read/write capability for a recently-created private file in `/tmp` is likely to have a much shorter lifetime. Thus, we could designate certain groups as volatile and others as stable, possibly with gradations in between, and assign capabilities to appropriate groups. Volatile groups would then become good candidates for low-cost invalidations.

Another way to minimize unintended revocations is to do background cleaning. When the metadata server is idle, it can help increase the number of available capability ID's by choosing one or more capability groups—preferably groups with many entries in the revocation list—and slowly migrating their valid capabilities to other groups. Migrating a capability means issuing an equivalent replacement capability in a different group and giving it out to clients that have the old capability; the clients replace the old capability with the new one. Once all valid capabilities in a group have been migrated, the metadata server can invalidate that group without causing any unintended revocations. Note that this scheme requires the metadata server to issue callbacks to the clients.

## 5 Performance

We ran experiments to evaluate the following: (1) the overhead of security, including MAC computation, capability revocation, and encryption; and, (2) system throughput and scalability under a bandwidth-intensive workload. Since the motivation of this work is to extend the performance benefits of NAD file systems to insecure environments, it is essential that the performance advantages of NAD file systems not be significantly reduced when security is added. We repeated each experiment using several different setups for comparison purposes.

The setups we used include the following: *non secure*, Snapdragon with all security turned off; *secure*, Snapdragon with access control, but without encryption; *private*, Snapdragon with access control and encryption; and, *NFS*, an NFS server with an attached local disk. Access control refers to the capability operations and replay detection needed to prevent unauthorized operations. Except where otherwise noted, encryption in this section refers to the encryption of all messages to and from the disk for privacy, and not to the encryption used for the client/server channel, which is part of Snapdragon's access control and hence present in both the secure and private setups. The non-secure setup does no MAC calculations, replay detection, capability operations, or encryption of any kind.

Our experiments were conducted on 3 to 8 Celeron 400 MHz PC's running Linux kernel version 2.4.12 and connected with a gigabit Ethernet switch. "Jumbo" 9,000-byte frames were enabled for network communications. Each machine has a locally-attached IDE disk with a maximum bandwidth of approximately 25 MB/second. In each experiment, one machine acts as the diskless metadata server, while others act as simulated disk controllers or diskless clients. (A simulated disk controller is the user-level program described in Section 3.1, which uses a raw disk partition as its backing store.)

A major difference between a real hardware NAD and our simulated one lies in the amount of memory available for the data cache. A commodity disk drive typically has a few megabytes, while the machines hosting our simulated NAD have 128 MB. Such a large cache would have a significant impact on NAD performance, because the disk controller could buffer and coalesce small random accesses into large sequential ones, improving the utilization of raw disk bandwidth.

Therefore, in order to make our simulated disk controllers more realistic, we limit their cache to 2 MB for these setups; that is, we force a sync to disk for ev-

ery 2 MB of dirty data that a simulated NAD receives; such scheme is appropriate for the streaming performance tests that we ran. In addition, we took the following measures to minimize the unintended effects of buffer caches: we freshly mounted the file systems and invalidated all block-device buffer caches before each experiment started, and flushed all buffer caches and unmounted the file systems before each experiment completed.

The capability scheme used in the experiments is the capability group method as described in Sections 2.2 and 3.4. But with the parameter values suggested there, group invalidations are very rare. To ensure the experiments included any performance implications of group invalidations, we used a much smaller store of capabilities—a strictly pessimistic alteration. Specifically, we set the number of groups ($g$) to 20, and the maximum number of capabilities in each group ($w_B$) to 500, allowing a maximum of 10,000 allocated capabilities. Therefore, for every 500 capabilities allocated beyond the first 10,000, a group needed to be invalidated.

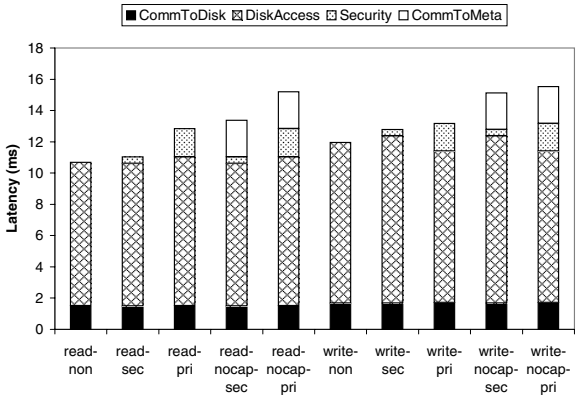


Figure 8: **Latency breakdown of read and write operations** for non-secure setup (non), secure setup (sec), and private setup (pri). Operations labeled with "nocap" means that the client does not have the appropriate capability and thus it has to request one to the metadata server. Latency is divided into the following categories: communication to the metadata server (CommToMeta), communication to the disk controller (CommToDisk), disk access, and security (including MAC computation and encryption).

## 5.1 Latency breakdown

We ran a set of micro benchmarks on Snapdragon and measured the latency of each operation in order to evaluate the various overheads associated with our security scheme. All the latency benchmarks were run on a collection of 700 files, each of size 4 KB. In each benchmark, a fixed filesystem operation (*e.g.*, read or chmod) was performed on each of the files in a randomized order. For the read and write cases, the metadata server, simulated NAD, and client driver were instrumented to report the time spent in fine-grained sub-operations.

Figure 8 shows the latency breakdown of the read and write operations with empty and synchronous write-through caches respectively. The physical disk access time averaged 9.3 ms for reads and 10.2 ms for writes. The MAC computation overhead was 0.4 ms and the encryption overhead was 1.4 ms. The disk-communication latency for all operations was 1.6 ms. If the client needs to request a capability for an operation, it requires an additional round trip from the client to the metadata server, which costs 2.3 ms. If a client attempts to use a revoked capability (not shown), it will get a rejection from the disk, which costs an extra 1.8 ms (secure setup) or 2.5 ms (private setup).

Figure 9 shows the latency of metadata operations. The chmod operation involves a round trip from the metadata server to the simulated NAD that requires MAC computation, while the unlink operation involves multiple such trips because ext2's unlink code writes multiple disk blocks. The open operation involves both MAC computation (to compute the secret $s$) and encryption of the capability, whether or not encryption for privacy is used. For operations that do not require revocations, the overhead for access control is less than 1 ms and the overhead for privacy is less than 3 ms. The operations involving revocations (*i.e.*, chmod-rev and unlink-rev) require an additional round-trip from the metadata server to the disk, roughly 1.4 ms.

In summary, access control (*i.e.*, MAC computation and replay detection) increases the latency of reads and writes by less than 0.5 ms (5%); encryption an additional overhead of 1.4 ms; and capability revocation increases the latency of read or writes latency by roughly 2.3 ms. For metadata operations, access control costs can cost 1 ms and privacy can cost 3 ms for certain operations.

## 5.2 Aggregate throughput and scalability

We ran a benchmark to measure the bandwidth of reads and writes by multiple clients on a single disk with various file sizes. There were 6 clients in our experiments, each running on a separate machine. Each client opened one file at a time and sequentially read or wrote in 64 KB chunks. Each experiment lasted between 5 and 25 min-

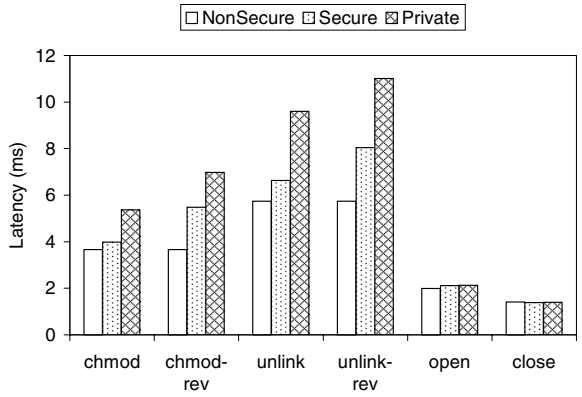


Figure 9: **Latency of metadata operations** under the non-secure, secure, and private setups. Operations labelled with "rev" require a capability revocation message to be sent from the metadata server to the disk.

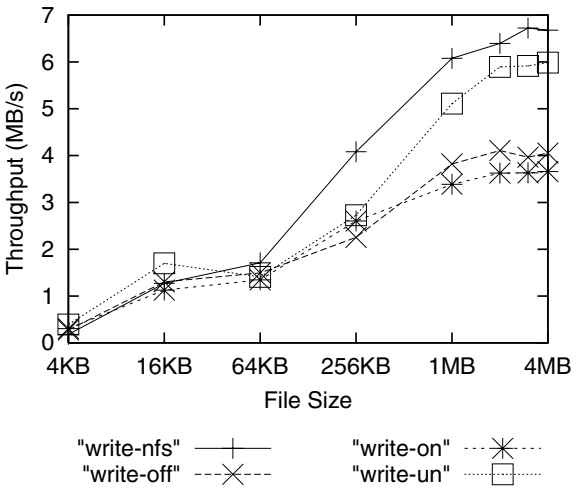


Figure 10: **Aggregate write bandwidth with 1 disk** and 6 clients for the secure setup (on), the non-secure setup (off), the NFS setup, which has no cache limit (nfs), and the non-secure setup modified to have no cache limit (un).

utes. All the files were stored on the same disk, but no file was accessed by more than one client or more than once during each experiment. All files within an experiment have the same size, but size varies from 4 KB to 4 MB between experiments.

The larger the file size, the less open/close overhead is incurred per transferred byte. There is also overhead associated with capability-group invalidation; the benchmarks using file sizes of 4 KB and 16 KB accessed more than 10,000 files and hence triggered group invalidation.

Figure 10 shows the system throughput as a function of file size for the write benchmarks. (The read benchmark results have similar trends and are not shown.) With file size 256 KB or less, the secure and non-secure setups have comparable bandwidth. With file sizes larger than 256 KB, the secure system performs up to 16% worse than the non-secure system. The difference is caused by CPU contention on the disk machine. Figure 11 shows the average percentage of idle time on the machine where the simulated NAD was hosted. The simulated disk controller in the secure setup consumes a considerable amount of cycles for MAC computation. Since it is implemented as a user-level process, it also consumes cycles for context switching and moving data across PCI buses and the kernel boundary.

We ran the same benchmark on an NFS server with a locally-attached disk (the NFS setup) for comparison. NFS performs comparably to Snapdragon (secure and non secure) for file sizes of 64 KB or less, and noticeably better for file sizes larger than 64 KB. This better performance is due to the NFS server's large data cache. Therefore, we ran the same benchmark again using the non-secure setup modified so that the simulated NADs can use as much cache as possible, up to the 128 MB physical memory capacity. The result is shown in Figures 10 and 11 as "write-un". The non-secure setup with no cache limit performed significantly better than the standard non-secure setup, which has only 2 MB of cache. This suggests that it would be worth increasing the data cache capacity in NADs (secure or non secure) in order to maximize bandwidth utilization for streaming I/O of large files by many concurrent users.

The idle time of the NFS server (shown in Figure 11) is not monotonic because the NFS server is performing both metadata and I/O operations. As the file size increases, the rate of metadata operations decreases, but the I/O rate increases.

We also ran a benchmark to measure the aggregate throughput for various numbers of disks and clients. Due to the limited number of machines available for our experiments, we had to collocate a client with a simulated NAD controller on each machine. Each client sequentially read or wrote files on a NAD hosted by another machine and each NAD was accessed by exactly 1 client. We ran the benchmark with 2 through 7 such machines. The file size was 256 KB and each client accessed 600 files in each run. Figure 12 shows the aggregate bandwidth as a function of the number of disks.

The results show that the aggregate read or write bandwidth of all clients scales linearly with the number of

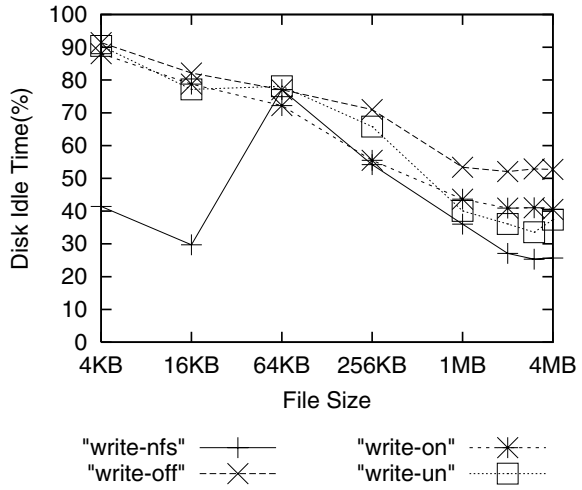


Figure 11: **Average percentage of idle CPU time on disk machines.**

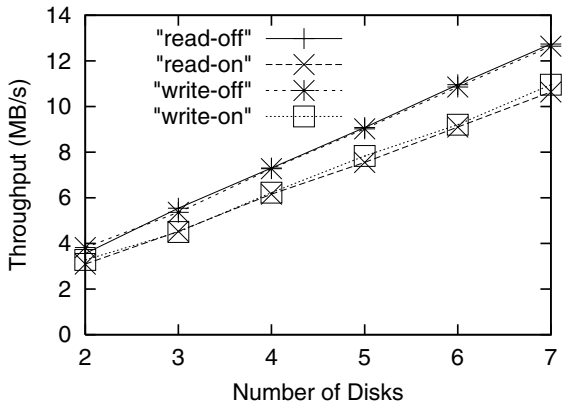


Figure 12: **Aggregate read/write bandwidth with multiple disks** for secure setup (on) and non-secure setup (off).

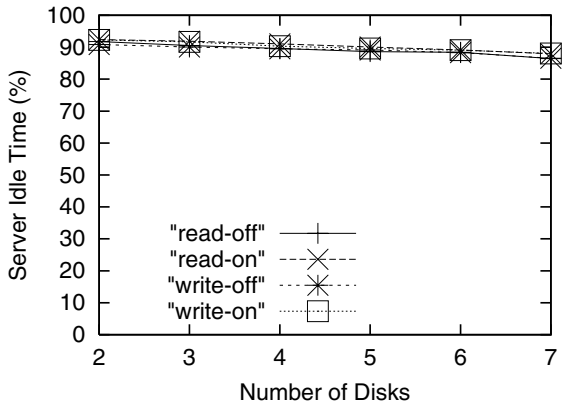


Figure 13: **Average percentage of idle CPU time on the metadata server.**

disks, which indicates that the metadata server imposes very low overhead to a high-bandwidth workload and has not become a bottleneck in a system with up to 7 disks. Figure 13 shows the average percentage of idle CPU time on the metadata server machine. The metadata-server machine was underloaded (*i.e.*, 86-92% idle) in these experiments. Therefore, we expect it to be able to support a considerably larger number of disks. The throughput of the non-secure setup grew faster than that of the secure setup because the access control overhead, which is dominated by MAC computation, is proportional to the data bandwidth.

## 5.3 Andrew benchmark with Linux kernel source

We ran a variant of the Andrew benchmark to show that Snapdragon has acceptable performance on a standard benchmark, even though Snapdragon was not designed for such workloads (*e.g.*, with extensive metadata operations and small files). Our variant of the Andrew benchmark differs only in that it uses as input the Linux kernel source, which contains 690 directories, 10,528 files and roughly 127 MB of data. Phase I of the Andrew benchmark duplicates the 690 directories 5 times in the file system being tested; phase II copies the files into one of the duplicated directories; phase III recursively lists all the duplicated directories; phase IV scans each copied file twice; and, phase V does a "make dep" and then "make" in the copied Linux kernel source directory, generating 1,362 new dependency and object files, or 13 MB of data.

The configuration for the Andrew benchmark includes only one client and NAD (or NFS server), each using separate machines. Figure 14 shows the elapsed time for each phase of Andrew benchmark for the secure, non-secure, and NFS setups.

In all phases, Snapdragon performed almost the same whether security was turned on or off, suggesting that the overhead of security is low. Snapdragon and NFS differed somewhat in phases I and II. The difference in phase I occurs because, for each new directory to be created, the Snapdragon metadata server needs to access the disk across the network, while the NFS server accesses the local disk directly. The difference in phase II is due to the overhead in opening and closing small files in Snapdragon—the Linux kernel source consists of mostly small files: 97% of the files are less than 64 KB,

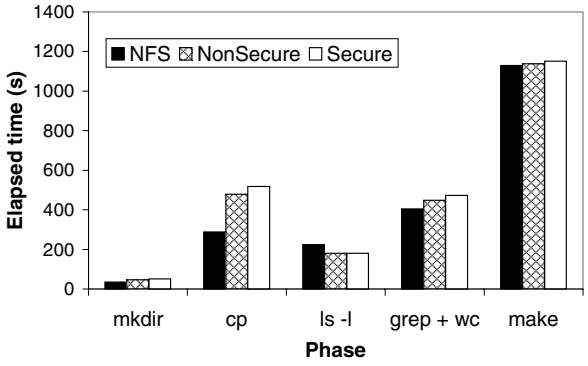


Figure 14: **Andrew benchmark with Linux kernel source.**

all but one file is less than 900 KB, and the largest file is roughly 2 MB. In phases III, IV and V, NFS and Snapdragon performed almost the same.

# 6 Limitations

One limitation of our block-based security approach is that we do not support files that are writable but not readable. This is because, under our scheme, to execute partial writes (writes to only part of a block), the client first needs to read the block's old contents—which requires read access—so that he can modify it. However, it is possible to overcome this problem by either having the disks support partial writes directly, or by having all such writes go through the metadata server.

We also do not support underlying file systems in which a block can contain data belonging to multiple files (*e.g.*, file systems in which the tails of many files are stored in a single block), because a block is the smallest unit of access control in our scheme. However, this problem can be overcome by changing capabilities so that they can optionally restrict access to a range of bytes within a block and by allowing disks to accept partial read and write requests.

Finally, with a log-based file system, it is not easy to exploit direct client access to disk when writing to the log, because accesses to the log need to be serialized. (This is not, however, a drawback of our security scheme, but rather a general limitation of asymmetric shared file systems with network-attached disks.) One possible solution is to make the disk the serialization point, but doing so would require adding considerable functionality to the disk.

# 7 Related work

NASD [7, 8, 10, 9] introduced the basic security architecture we use, where a central server decides policy and the disks implement only a simple access mechanism based on cryptographic capabilities. We differ in our handling of revocations and replay attacks, as described in Sections 2.2 and 2.4, and in the fact that our capabilities specify permission in terms of ranges of physical blocks rather than object IDs. SCARED [19] generalizes the NASD security framework to allow for identity-based access, where the client proves its identity to the disk, which then decides what access should be allowed based on its understanding of file access-control lists (ACLs). We prefer a capability-based scheme, however, because it does not require restrictions on the disk layout so that the disk can decode ACLs.

NASD's network-attached disks are complicated pieces of machinery, possessing most of the functionality of a file server: rather than simply serving raw blocks as our NADs do, they present the abstraction of a collection of numbered, but unnamed, variable-size data objects, which are byte sequences with a small number of attributes. A central server manages a collection of NASD disks, providing clients with the usual illusion of a hierarchical file system. This is done (in the absence of striping) by mapping each directory or data file to a single NASD object.

SUNDR [13] and SNAD [5, 15] do away with the central server altogether and use more powerful cryptographic methods (*e.g.*, blocks on disk are encrypted and "signed" using keys known only to clients) to stop an attacker that can control disks from reading or undetectably altering user data. Most of their complexity is due to this stronger security level, which is not useful for the scenarios that we envision, where the people most likely to be able to compromise servers or disks (*i.e.*, the system administrators) would also easily be able to compromise clients, defeating these systems' security.

SUNDR and SNAD use the *encrypt-on-disk* strategy, where data is concealed by keeping it encrypted on the disk. Revoking access in such systems is expensive because the involved data must be re-encrypted using a new key. Riedel *et al.* [21] argue that encrypt-on-disk may offer better performance when privacy is desired than *encrypt-on-wire* systems such as ours, because encrypt-on-disk encrypts and decrypts data only at the client, not at both the client and disk.

Although encrypt-on-disk leaks more information to eavesdroppers (blocks can be identified on the wire because they are re-encrypted only when rewritten, in-

stead of each time they are transmitted), should the extra encryption prove burdensome we think it is possible to modify Snapdragon to get the performance benefit (but not the security benefit) of encrypt-on-disk without changing our disk protocol.[5]

The Netstation project sketches how their *Derived Virtual Device* (DVD) abstraction, a very general mechanism for securely delegating access to an arbitrary subset of a network-attached device, can be used to create a block-oriented secure NAD file system called STORM [26]. Rather than use capabilities for security, they use Kerberos authentication to authenticate client requests, which specify a DVD ID. Devices maintain a table of which DVDs each client is allowed to access as well as detailed information about the access restrictions of each DVD. While the notion of DVDs is very elegant, it may be too inefficient to be of use in a production file system: the DVD access information (largely blockmap information for STORM) requires extra network trips to be installed by the server, uses a lot of disk memory, and seems to be installed by downloading functions written in Scheme.

We believe our paper is the first one to use Bloom filters to protect against replay attacks. However, Bloom filters have long existed and have other uses. Superficially related to our paper is the work of [6], which uses Bloom filters for revocations (instead of replay attacks).

# 8 Conclusion

In this paper we have presented a new block-based security scheme for network-attached disks (NADs). In contrast to previous work, our scheme requires no changes to the data layout on disk and only minor changes to the standard protocol for accessing remote block-based devices. Thus, existing NAD file systems and storage-management software could incorporate our new secure NADs with only incremental changes. Moreover, our scheme's demands on the NADs are modest: standard cryptographic functionality plus very little RAM. The low need for RAM is achieved by two novel features: our revocation scheme based on capability groups, and a replay-detection method using Bloom filters. We believe our design could be easily deployed in existing NAD's or in disk arrays with minimal changes.

We implemented a prototype secure NAD file system using our scheme, and evaluated its performance and scalability. The cost of access control is small: Latency for reads and writes increases by less than 0.5 ms (5%), and the bandwidth decreases by up to 16%. The system throughput scales linearly with the number of disks supported by a single metadata server (up to 7 in our experiments).

Hence, we believe our scheme is a practical and efficient method for incorporating security into existing NADs with minimal change—a scheme that could liberate NAD file systems from the confines of the machine room and data center, allowing them to reach a broader range of users directly, yet securely.

[5] We would store blocks encrypted on disk, but the keys would be managed by the metadata servers.

# Acknowledgments

We would like to thank Fay Chang and John Wilkes for useful discussions as well as our shepherd Greg Ganger and the anonymous referees for helpful suggestions.

# References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *Lecture Notes in CS*, 1109:1–15, 1996.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[4] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970.

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

[6] Eran Gabber and Abraham Silberschatz. Agora: A minimal distributed protocol for electronic commerce. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 223–232, 1996.

[7] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks. Technical Report CMU–CS–97–112, Carnegie Mellon, March 1997.

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

[9] Howard Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, CMU, 1999.

[10] Howard Gobioff, Garth Gibson, and Doug Tygar. Security for network attached storage devices. Technical Report CMU–CS–97–185, Carnegie Mellon, October 1997.

[11] Kent Koeninger. CXFS: A clustered SAN filesystem from SGI. http://www.sgi.com/Products/PDF/2691.pdf.

[12] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.

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

[14] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Trans. on Computer Systems*, 2(3):181–197, 1984.

[15] Ethan L. Miller, William E. Freeman, Darrell D. E. Long, and Benjamin C. Reed. Strong security for network-attached storage. In *Proceedings of the FAST 2002 Conference on File And Storage Technologies*, pages 1–14, January 2002.

[16] S. Mittra and T. Woo. A flow-based approach to datagram security. In *Proc. ACM SIGCOMM*, 1997.

[17] NIST. Secure hash algorithm, 1995. FIPS 180-1.

[18] Matthew T. O'Keefe. Shared file systems and Fibre Channel. In *Proceedings of the Sixth NASA Goddard Conference on Mass Storage Systems*, pages 1–16. IEEE Computer Society Press, 1998.

[19] B. Reed, E. Chron, D. Long, and R. Burns. Authenticating network attached storage. *IEEE Micro*, 20(1), January 2000.

[20] Hans Reiser. Reiserfs v.3 whitepaper, 2000. http://www.namesys.com/.

[21] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)*, pages 15–30, January 2002.

[22] R. Sandber, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of USENIX Summer Conference*, 1985.

[23] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[24] V. Srinivasan and J. C. Mogul. Spritely NFS: Experiements with cache-consistency protocols. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 45–57, December 1989.

[25] Tivoli. Tivoli SANergy: Helping you reach your full SAN potential. http://www.tivoli.com/-products/documents/datasheets/sanergy_ds.pdf.

[26] Rodney Van Meter, Gregory Finn, and Steven Hotz. Derived virtual devices: A secure distributed file system mechanism. In Ben Kobler, editor, *Proc. Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, September 1996.