Design and Implementation of a Self-Securing Storage Device

John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz,

Craig A.N. Soules, Gregory R. Ganger

May 2000

CMU-CS-00-129

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

Abstract

Self-securing storage prevents intruders from undetectably tampering with or permanently deleting stored data. To accomplish this, self-securing storage devices internally audit all requests and keep all versions of all data for a window of time, regardless of the commands received from potentially compromised host operating systems. Within the window, system administrators are guaranteed to have this valuable information for intrusion diagnosis and recovery. The S4 implementation combines log-structuring with novel metadata journaling and data replication techniques to minimize the performance costs of comprehensive versioning. Experiments show that self-securing storage devices can deliver performance that is comparable with conventional storage. Further, analyses indicate that several weeks worth of all versions can reasonably be kept on state-of-the-art disks, especially when differencing and compression technologies are employed.

The authors would like to thank the member companies of the Parallel Data Consortium (CLARiiON Array Development, EMC Corporation, Hewlett-Packard Labs, Hitachi, Infineon Technologies, Intel Corporation, LSI Logic, MTI Technology Corporation, Novell, Inc., PANASAS, L.L.C., Procom Technology, Quantum Corporation, Seagate Technology, Sun Microsystems, Veritas Software Corporation, and 3Com Corporation). The authors also thank IBM Corporation and CMU's Data Storage Systems Center for their support of this project.

Keywords: Security, survivability, intrusion tolerance, storage systems, network-attached storage

1 Introduction

Despite the best efforts of system designers and implementors, it has proven difficult to prevent computer security breaches. This fact is of growing importance as organizations find themselves increasingly dependent on wide-area networking (providing more potential sources of intrusions) and computer-maintained information (raising the significance of potential damage). A successful intruder can obtain the rights and identity of a legitimate user or administrator. With these rights, it is possible to disrupt the system by accessing, modifying, or destroying critical data.

Even after an intrusion has been detected and terminated, system administrators face two difficult tasks: determining the damage caused by the intrusion and restoring the system to a safe state. Damage includes compromised secrets, creation of back doors and Trojan horses, and tainting of stored data. Detecting each of these is made difficult by crafty intruders who understand how to scrub audit logs and disrupt automated tamper detection systems. System restoration involves identifying a clean back-up (i.e., one created prior to the intrusion), reinitializing the system, and restoring information from the back-up. Such restoration often requires a significant amount of time, reduces the availability of the original system, and frequently causes loss of data created between the safe back-up and the intrusion.

Self-securing storage offers a partial solution to these problems by preventing intruders from undetectably tampering with or permanently deleting stored data. Since intruders can take the identity of real users and even the host OS, any resource controlled by the operating system is vulnerable, including the raw storage. Rather than acting as slaves to host operating systems, self-securing storage devices view them, and their users, as questionable entities for which they work. These self-contained, self-controlling devices internally version all data and audit all requests for a guaranteed amount of time (e.g., a week), thus providing system administrators time to detect and recover from intrusions. The critical difference between self-securing storage and host-controlled versioning (e.g. Elephant [29]) is that intruders can no longer bypass the versioning software by compromising a complex OS or its poorly-protected user accounts. Instead, intruders must compromise single-purpose devices that export only a simple storage interface, and in some configurations, they may have to

compromise both.

This paper describes self-securing storage and our implementation of a self-securing storage system, called S4. A number of challenges arise when storage devices distrust their clients. Most importantly, it may be difficult to keep all versions of all data for an extended period of time, and it is not acceptable to trust the client to specify what is important to keep. Fortunately, disk capacities increase faster than most computer characteristics (100%+ per annum in recent years). Analysis of recent workload studies suggests that it is possible to version all data on modern 30-100GB drives for far longer than a week [29, 35]. Further, aggressive compression and cross-version differencing techniques extend the intrusion detection window offered by self-securing storage devices. Other challenges include maintaining on-disk locality when blocks cannot be overwritten, achieving secure administrative control, and dealing with denial-of-service attacks.

The S4 system addresses these challenges with a new storage management structure. The storage management system uses a log-structured object system for data versions, a novel journal-based structure for metadata versions, and an opportunistic on-disk *anti-entropy cache* for restoring sequentiality to version-scrambled objects. In addition to reducing space utilization, the metadata journaling simplifies background compaction and reorganization for blocks shared across many versions. Experiments with S4 show that the security and data survivability benefits of self-securing storage can be realized with reasonable performance. Specifically, the performance of network-attached S4/NFS is comparable to FreeBSD's NFS for both microbenchmarks and application benchmarks.

The remainder of this paper is organized as follows. Section 2 discusses intrusion survival and recovery difficulties in greater detail. Section 3 describes how self-securing storage addresses these issues, presents some challenges inherent to self-securing storage, and discusses design solutions for addressing them. Section 4 describes the implementation of S4. Section 5 evaluates the S4 implementation. Section 6 discusses a number of open issues related to self-securing storage. Section 7 discusses related work. Section 8 summarizes this paper's contributions.

2 Intrusion Diagnosis and Recovery

Upon gaining access to a system, an intruder has several avenues of mischief. Most intruders attempt to destroy evidence of their presence by erasing or modifying system log files. Many intruders also install back doors in the system, allowing them to gain entry at will in the future. They may also install other software, read and modify sensitive files, or use the system as a platform to launch additional attacks. Depending on the skill with which the intruder hides his presence, there will be some *detection latency* before the intrusion is discovered by an automated intrusion detection system (IDS) or by a suspicious user or administrator. During this time, the intruder can continue his malicious activities while users continue to use the system, thus entangling legitimate changes with those of the intruder. Once an intrusion has been detected and discontinued, the system administrator is left with two difficult tasks: diagnosis and recovery.

Diagnosis is challenging because intruders can usually compromise the "superuser" account on most operating systems, giving them full control over all resources. In particular, this gives them the ability to manipulate everything stored on the system's disks, including audit logs, file modification times, and tamper detection utilities. Recovery is difficult both because diagnosis is difficult and because user-convenience is an important issue. This section discusses intrusion diagnosis and recovery in greater detail, and the next section describes how self-securing storage addresses these issues.

2.1 Diagnosis

Intrusion diagnosis consists of three phases: detecting the intrusion, discovering the weaknesses that were exploited (for future prevention), and determining what the intruder has done. All are difficult when the intruder has free reign over storage and the OS.

Without the ability to protect storage from compromised operating systems, intrusion detection may be limited to attentive users and system administrators noticing odd behavior. Examining the system logs is the most common approach to intrusion detection [6], but when intruders can manipulate the log files, such an approach is not useful. Some intrusion detection systems also look for changes to important system files [17]. These systems are

vulnerable to intruders who can change what the IDS thinks is a“safe” copy.

Determining how the intruder compromised the system is often impossible in conventional systems, because he will scrub the system logs. In addition, any tools that may have been stored on the target machine for use in multi-stage intrusions may have been deleted. The common “solutions” are to try to catch the intruder in the act or to hope that he forgot to delete his exploit tools.

The last step of diagnosing an intrusion is to discover what was accessed and modified by the intruder. This is extremely difficult, because file access and modification times can be changed, and system log files can be doctored. In addition, checksum databases are of limited use, since they are effective only for static files, thus providing no protection for user data.

2.2 Recovery

Because it is usually not possible to diagnose an intruder’s activities, full system recovery generally requires that the compromised machine be wiped clean and reinstalled from scratch. Prior to erasing the entire state of the system, users may insist that critical data be saved. Critical data is any data that has changed since the last backup and requires significant effort to recreate. The more effort that went into creating the changes, the more motivation there is to keep this data. Unfortunately, as the size and complexity of the data grows, the likelihood that tampering will go unnoticed increases. Foolproof assessment of the data is very difficult, and overlooked modifications may hide tainted information or a back door inserted by the intruder.

Upon restoring the OS and any applications on the system, the administrator must identify a backup that was made prior to the intrusion; the most recent backup may not be usable. After restoring data from a verified backup, the critical data can be restored to the system, and users may resume using the system. This process often takes a considerable amount of time—time during which users are denied service.

3 Self-Securing Storage

Self-securing storage ensures information survival and auditing of all accesses by establishing a security perimeter around the storage device. Conventional storage devices are slaves to the host operating system, relying on it for protection of the users' data. A self-securing storage device operates as an independent entity, tasked with the responsibility to not only store data, but to protect it as well. This shift of storage security functionality into the storage device's firmware allows data and audit information to be safeguarded in the presence of file server and client system intrusions. Even if the OSes of these systems are compromised and an intruder is able to issue commands directly to the self-securing storage device, the new security perimeter remains intact.

Behind the security perimeter, the storage device ensures data survival by keeping all versions of data. This *history pool* of old data versions, combined with the audit log of accesses, can be used to diagnose and recover from intrusions. This section discusses the benefits of self-securing storage and several core design issues that arise in realizing it.

3.1 Enabling intrusion survival

Self-securing storage assists in intrusion recovery by allowing the administrator to view audit information and quickly restore modified or deleted files. The audit logs of data accesses help to diagnose intrusions and detect the propagation of any maliciously modified data.

Self-securing storage maintains old versions of data objects. This simplifies diagnosis of an intrusion since system logs and programs cannot be imperceptibly altered. Because of this, self securing storage makes conventional tamper detection systems obsolete. In addition, since the drive maintains these old versions, they can quickly be restored to their pre-intrusion state.

Since the administrator has the complete picture of the system's state, from intrusion until discovery, it is considerably easier to establish the method used to gain entry. For instance, the system logs would have normally been erased, but by examining the versioned copies of the logs, the administrator can see any messages that were generated during the intrusion and later removed. In addition, any exploit tools temporarily stored on the system

may be recovered.

Previous versions of system files, from before the intrusion, can be quickly and easily restored by resurrecting them from the history pool. This prevents the need for a complete re-installation of the operating system, and it does not rely on having a recent, off-line backup or up-to-date checksums (for tamper detection) of system files. Additionally, by utilizing the storage device's audit log, it is possible to assess which data might have been directly affected by the intruder. Further, there is no need to pre-back-up "critical files" before the restore, since files can be selectively pulled forward by the user and/or backed up after restoration.

The data protection provided by self-securing storage allows easy detection of modifications, selective recovery of tampered files, prevention of data loss due to out-of-date backups, and speedy recovery, since data need not be loaded from an off-line archive.

3.2 Device security perimeter

The device's security model is what makes the ability to keep old versions more than just a user convenience. The security perimeter consists of self-contained software that supports only a simple storage interface to the outside world and verifies each command's integrity before processing it. In contrast, most file servers and client machines run a multitude of services that are susceptible to attack. Since the self-securing storage device is a single-function embedded device, the task of making it secure is much easier; compromising its firmware is analogous to breaking into an IDE or SCSI disk.

For network-attached devices (as compared to devices attached directly to a single host system), the internally managed audit log becomes more useful if the device can verify each request as coming from both a valid user and a valid client. This can allow the device to enforce access control decisions and partially track propagation of tainted data. If clients must be authenticated, requests can be tracked to a single client machine, and the device's audit log can yield the scope of direct damage from the intrusion of a given machine. By making sure any given request is bound to a {client, user} pair, a self-securing storage device can assure the following:

- For an uncompromised client, requests are bound to the correct user's credentials and not those of another user on that machine. Any client not exhibiting this behavior would be considered compromised.

- For a compromised client, accesses are bound to the correct machine's credentials, but user information may or may not be correct.

Network-attached storage must also deal with privacy and authenticity of network traffic [7, 9]. One solution would be the use of a network-level mechanism like IPSec [16], for which hardware support is expected to minimize the performance consequences.

3.3 History pool management

The old versions of objects kept on the drive comprise the history pool. Every time an object is modified or deleted, the version that existed just prior to the modification becomes part of the history pool. Eventually the previous version will age and have its space reclaimed by the drive. Because clients cannot be trusted to demarcate versions consisting of multiple modifications, a separate version must be kept for every modification. This is in contrast to versioning file systems that generally create new versions only when a file is closed.

A self-securing storage device guarantees a lower bound on the amount of time that a deprecated object remains in the history pool before it is reclaimed. During this window of time, the old version of the object can be completely restored by requesting that the drive *copy forward* the old version, thus making a new version. The window of time during which an object can be restored is called the *detection window*. When determining the size of this window, the administrator must examine the tradeoff between the detection latency provided by a large window and the extra disk space that is consumed by the proportionally larger history pool.

While the capacity of disk drives is growing at an incredible rate, it is still finite, which presents two problems:

1. Providing a reasonable detection window in exceptionally busy systems.

2. Dealing with malicious users that attempt to fill the history pool. (Note that space

exhaustion attacks are not unique to self-securing storage. However, device-managed versioning makes per-user quotas ineffective for limiting them.)

In a busy system, the amount of data written could make providing a reasonable detection window difficult. Fortunately, the analysis in section 5.2 suggests that multi-week detection windows can be provided in many environments at a reasonable cost. Further, aggressive compression and differencing of old versions can significantly extend the detection window.

Deliberate attempts to overflow the history pool cannot be prevented by simply increasing the space available, and as with most denial of service attacks, there is no perfect solution. There are three flawed approaches to addressing this type of abuse. The first is to have the device reclaim the space held by the oldest objects when the history pool is full. Unfortunately, this would allow an intruder to destroy information by causing its previous instances to be reclaimed from the overflowing history pool. The second flawed approach is to stop versioning objects when the history pool fills; while this will allow recovery of the old data, system administrators would no longer be able to diagnose the actions of an intruder or differentiate them from subsequent legitimate changes. The third approach is for the drive to deny any action that would require additional versions once the history pool fills; this would result in denial of service to all users (legitimate or not).

Our hybrid approach to this problem is to try to prevent the history pool from being filled by detecting probable abuses and throttling the source machine's accesses. When successful, this allows human intervention before the system is forced to choose from the above poor alternatives. Selectively increasing latency and/or decreasing bandwidth allows well-behaved users to continue to utilize the system even while it is under attack. Experience will show how well this works in practice.

Since the history pool will be used for intrusion diagnosis and recovery, not just recovering from accidental destruction of data, it is difficult to construct an algorithm that would save space in the history pool by pruning versions within the detection window. Almost any algorithm that could be constructed to selectively remove versions has the potential to be abused by an intruder to cover his tracks and to successfully destroy/modify information during a break-in.

3.4 Interface to history information

The history pool contains a wealth of information about the system's recent activity. This makes accessing the history pool a sensitive operation, since it allows the resurrection of deleted and overwritten objects. This is a standard problem posed by versioning file systems, and it is exacerbated by the inability to selectively delete versions.

There are two basic approaches that can be taken toward access control for the history pool. The first is to allow only a single administrative entity to have the power to view and restore items from the history pool. This could be useful in situations where the old data is considered to be highly sensitive. Having a single tightly-controlled key for accessing historical data decreases the likelihood of an intruder gaining access to it. While this improves security, it prevents users from being able to recover from their own mistakes, thus consuming the administrator's time to restore users' files. The second approach is to allow users to recover their own old objects (in addition to the administrator). This provides the convenience of a user being able to recover their deleted data easily, but also allows an intruder, who obtains valid credentials for a given user, to recover that user's old file versions. It is important to note that permitting full deletion of objects would be perilous to the integrity of the data, since such a mechanism could be used by intruders to destroy information.

Our compromise is to allow users to selectively decide, on a file by file basis. By choice, a user could thus delete an object, version, or all versions from visibility by anyone other than the administrator. Complete removal should not be permitted, since permanent deletion of data via any other method than aging would be unsafe. This would allow users to enjoy the benefits of versioning for presentations and source code, while preventing access to visible versions of embarrassing images or unsent e-mail drafts.

3.5 Version-administration tools

Since self-securing storage devices store versions of raw data, users and administrators will need assistance in parsing the history pool. Tools for traversing the history must assist by bridging the gap between standard file interfaces and the raw object versions that are stored on the device. By being aware of both the versioning system and formats of the data objects,

utilities can present interfaces similar to that of Elephant [29], with "time-enhanced" versions of standard utilities such as `ls` and `cp`.

In addition to allowing a simple view of data objects in isolation, intrusion diagnosis tools can utilize the audit log to provide an estimate of damage. For instance, it is possible to see all files and directories that a client modified during the period of time that it was compromised. Further estimates of the propagation of data written by compromised clients are also possible, though imperfect. For example, diagnosis tools may be able to establish a link between objects based on the fact that one was read just before another is written. Such a link between a `*.c` source file and its corresponding `*.o` would be useful if a user determines that a source file had been tampered with; in this situation, the object file should also be restored or removed. Exploration of such tools will be an important area of future work.

4 Implementation of the S4 device

S4 is a self-securing storage device, which maintains an efficient object-versioning system transparently for its clients. It aims to perform comparably with current systems, while providing the benefits of self-securing storage and minimizing the corresponding space explosion.

4.1 Object store

Considerable research has gone into providing a useful abstraction above the standard block-level interface to mass storage devices. This work resulted in several proposals and prototypes for object-based disks [8, 25]. It has also shown that the object abstraction simplifies access control by treating an arbitrary collection of bytes as a management unit, as compared to a standard block device. Since the S4 drive is responsible for enforcing and managing its own access control decisions, we have chosen an object-based interface.

In S4, objects exist in a flat namespace managed by the drive. When objects are created, they are given a unique identifier (`ObjectID`) by the drive which is used by the client for all future references to that object. Each object has an access control structure to determine

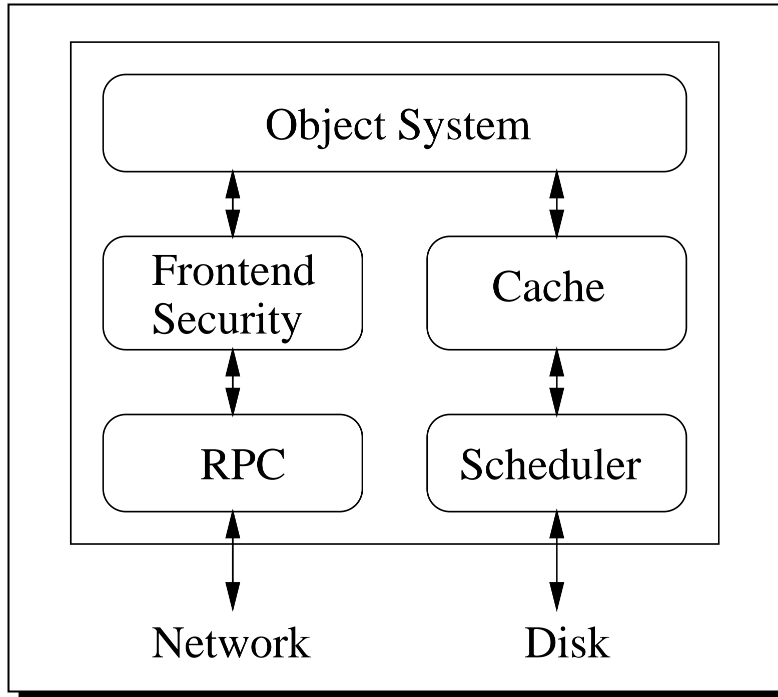

Figure 1: **S4 Drive Overview.**

which entities (users and client machines) have permission to access the object. Objects also have metadata, file data, and an opaque attribute storage space (for use by client file systems) associated with them.

The S4 object system is layered above two separate subsystems: the infrastructure and the front end (see Figure 1). The infrastructure contains the drive's RPC and disk interface, and exports all of the drive's interfaces to clients through a set of RPCs (see Table 1) layered over the network [24, 33]. The front end acts as a communication layer between the RPC and the object store, and enforces security.

To support persistent mount points, an S4 drive uses "named objects." The object names are associations of arbitrary ASCII strings with particular `ObjectID`s. The table of named objects is implemented as a special S4 object accessed through dedicated partition manipulation RPC calls. Since this table is implemented as an object, it is versioned in the same manner as all objects on the S4 drive.

RPC Type	Description
`Create`	Create an object
`Delete`	Delete an object
`Sync`	Sync entire cache to disk
`SyncO`	Sync object to disk
`Flush`	Removes all versions of all objects before a specified time
`FlushO`	Removes all versions of an object before a specified time
`Truncate`	Truncate an object
`Read`	Read data from an object
`Write`	Write data to an object
`Append`	Append data to end of an object
`GetAttr`	Get attributes of an object
`SetAttr`	Set attributes of an object
`GetACLByUser`	Gets an ACL associated with a UID for an object
`GetACLByIndex`	Gets an ACL by its index in the ACL table
`SetACL`	Set the ACL of an object
`PCreate`	Create a partition
`PDelete`	Delete a partition
`PList`	List partitions
`PMount`	Mount a partition

Table 1: **S4 Remote Procedure Call List.** Note that all modifications create new versions without effecting the previous version.

4.2 On-disk data organization

The main goals for the S4 object system are to avoid performance overhead and to minimize wasted space, while keeping all versions of all objects for a given period of time. Achieving these goals required a combination of known and novel techniques for organizing on-disk data.

Since data within the history pool cannot be overwritten, the object system uses a log structure similar to LFS [28]. This structure allows multiple data and metadata updates to be clustered into fewer, larger writes and obviates any need to move previous versions before writing. With additional metadata, the old versions can be retrieved. To do this efficiently, we use a variation of journaling to track object changes. Marking changes with

journal entries saves space and simplifies the process of recreating old object versions. A segment cleaner, similar to LFS's cleaner, removes expired versions from the history pool.

Several studies have examined log-structured file systems' performance problems and ways to alleviate them [21, 30]. One of the bigger problems is the loss of data locality for frequently changing files. We have created a new data structure called an *anti-entropy* cache to help solve this problem. An anti-entropy cache opportunistically keeps an additional read-optimized copy of the object on the disk. If the drive is low on space, this copy can be reclaimed to allow for further log growth.

4.2.1 Object structure

An S4 disk object consists of two basic parts, an *onode* and an *ACL table*. The *onode* holds standard inode metadata along with an opaque space for file system specific attributes. S4-specific metadata includes the size, create and modification times, and the direct and indirect block pointers. The additional attribute space makes it possible to implement a variety of file systems on top of S4's object interface. An onode has 30 direct block pointers and one single, one double, and one triple indirect block pointer which provides a maximum object size of just over 32GB.

The *ACL table* holds the access control list for the object. The table contains 64 {user, access rights} pairs, which can be expanded to a special ACL object if the table grows beyond this limit. Pairing the *onode* and *ACL table* together on the disk allows for efficient ACL access in the average case, without eliminating the possibility of large access control lists.

4.2.2 Overall disk layout

The S4 Object system divides the disk into fixed size log segments. LFS used a segment size of 1MB, but based on recent research results [21], we chose a segment size of 64KB. We group these segments into 4GB allocation groups, which contain a summary segment marking each 4GB boundary (see Figure 2). This *summary segment* contains a copy of the superblock, the free segment bitmap for that allocation group, and indirect blocks used for an object map, similar to LFS's inode map. S4 uses the object map to track the location of the most current copy of an object's metadata within the log, and potentially, its anti-entropy cache.

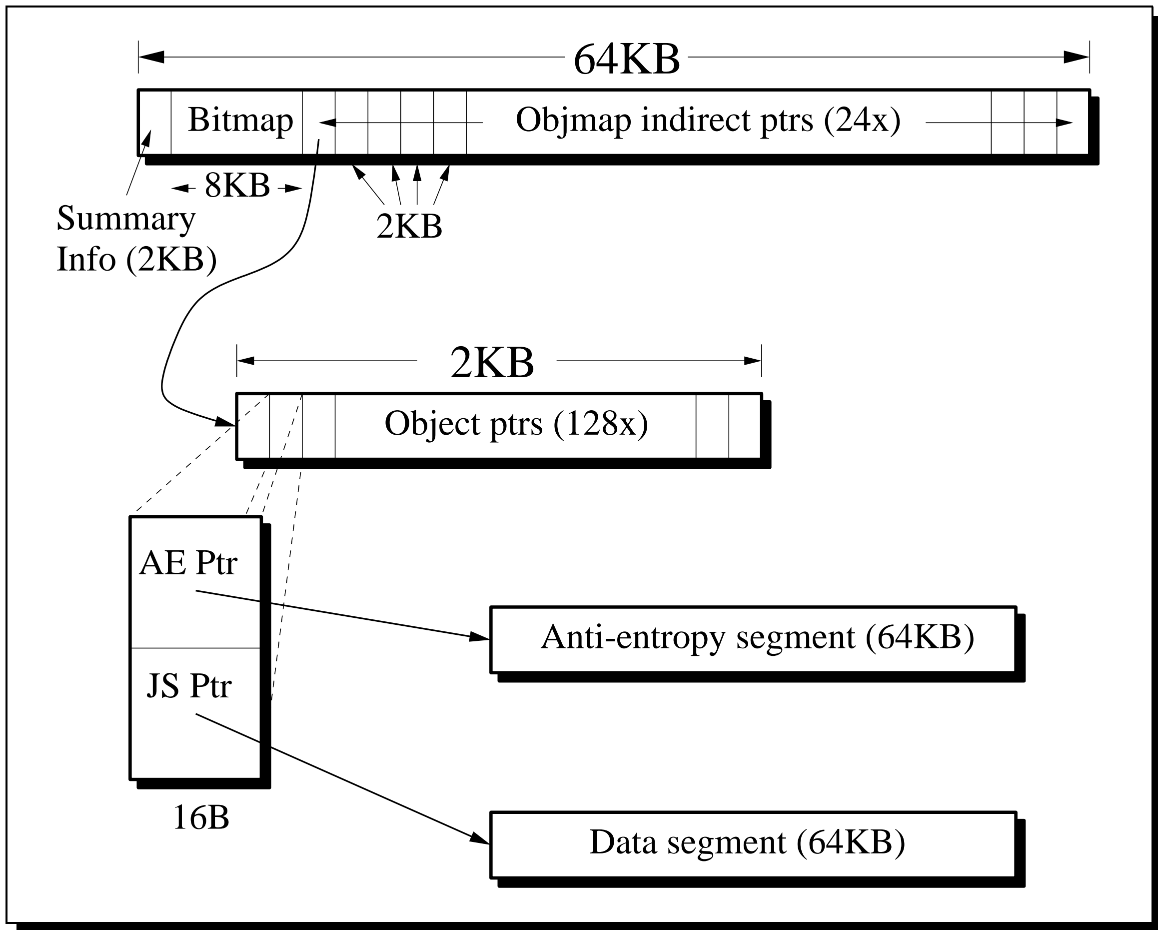

Figure 2: **Disk layout overview.** Shows the allocation group summary segment and the object map.

This flexible structure gives us the ability to have a large number of objects in the system without having to pre-allocate space for them.

S4 does all of its log writes to data segments (see Figure 3). Each *data segment* has one block for summary information and 31 blocks available for journal entries and data. The summary block keeps track of the layout and allocation of space within the segment. Specifically, it has a free count and bitmap to track available space, and a mapping of objects to their metadata within the segment.

4.2.3 Journal-based metadata versioning

S4 stores the history of metadata changes in a journal. Because clients are not trusted to notify S4 when objects are closed, every update requires a new version and thus a new onode. Further, when data pointed to by indirect blocks are modified, the indirect blocks

must be versioned as well. For triple-indirect blocks, a single-block update could require 4 new blocks and a new onode. Early experiments with such a conventional versioning system showed that writing a large file could cause up to a 4x growth in disk usage. Conventional versioning file systems avoid this performance problem by only creating new versions when a file is closed.

S4's journal-based approach significantly reduces these problems. By persistently keeping journal entries of all metadata changes, metadata writes can be safely delayed, since onodes and indirect blocks can be recreated in the event of a failure. To avoid rebuilding an object's current state from the journal during normal operation, an object's onode and ACL table are committed to the log before the object is evicted from the cache. For the same reasons, indirect block writes can also be delayed until this time, since the journal entries contain sufficient data to recreate them.

S4 uses six kinds of journal entries: `attribute`, `ACL`, `delete`, `truncate`, `write`, and `checkpoint`. Each of these entries represent a specific metadata change with the exception of `checkpoint`. Checkpointing denotes writing a consistent copy of the onode, ACL table, and indirect blocks of an object into the log. Also, all entries can serve to either undo or redo the specified operation. The undo operation is necessary to allow in-time access of objects, while the redo is helpful for the cleaner, allowing it to efficiently roll changes forward as it deletes expired versions.

Storing an object's changes within the log is done using *journal sectors*. Each journal sector contains the packed journal entries that refer to a single object's changes made within that data segment (see Figure 4). The sectors are tracked within a data segment using the metadata mapping found in the summary information. Journal sectors are chained together backward in time to allow for version reconstruction.

Journal entries also allow efficient differencing between versions. Since the exact changes between writes are noted within the entry, it is easy to find the blocks that should be compared. Once the differencing is complete, the old blocks can be discarded, and the difference left in its place. For subsequent reads of old versions, the data for each block must be recreated as the entries are traversed. Still, cross-version differencing of old data will often be effective in reducing the amount of space used by old versions [2].

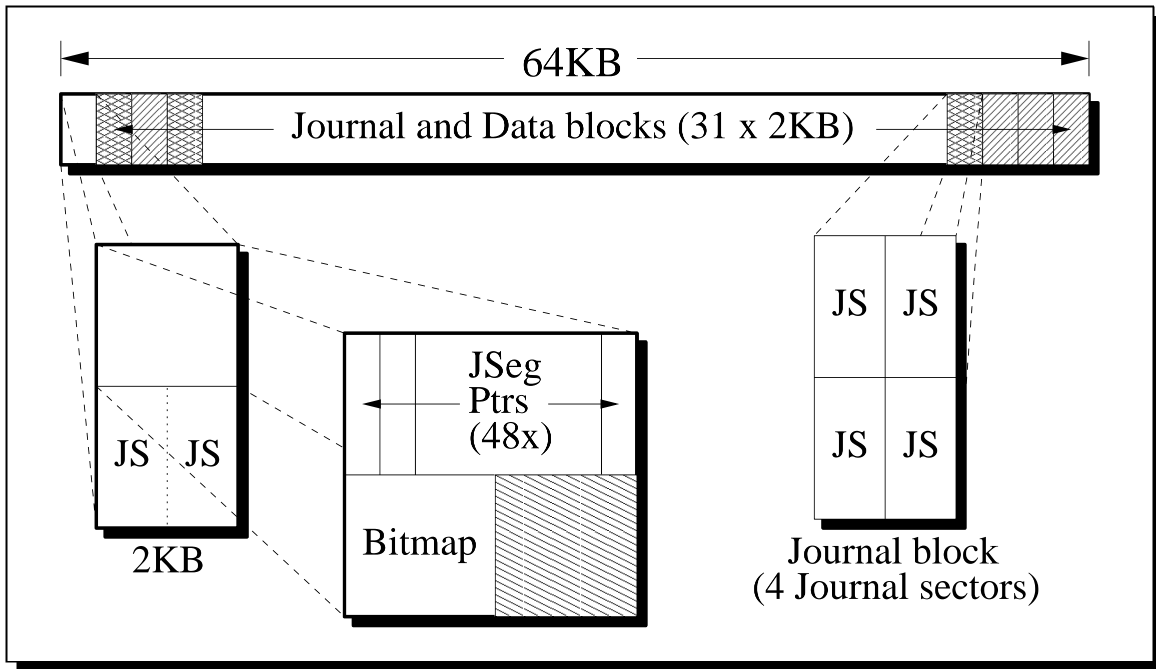

Figure 3: **Data Segment.** This structure holds all log data.

4.2.4 Anti-entropy cache

The *anti-entropy cache* keeps a sequential copy of an object's direct blocks (up to 60KB) along with the object's onode and ACL table in a special segment. When a read request is made, the entire anti-entropy cache can be read, retrieving the object's metadata along with the stored data in one, quick operation. Because of their limited size, they currently have little effect on large files, but smaller, rapidly changing files can have improved read performance over other log-structured file systems.

The anti-entropy caches are organized into *anti-entropy segments*. As shown in Figure 5, an anti-entropy segment can contain multiple anti-entropy caches. When a cache grows beyond the available space within its segment it must be moved into a new segment. Since anti-entropy caches are simply a read optimization, they can be freed easily if disk space is immediately needed. Also, since they are created in the background, they have little performance penalty for other operations.

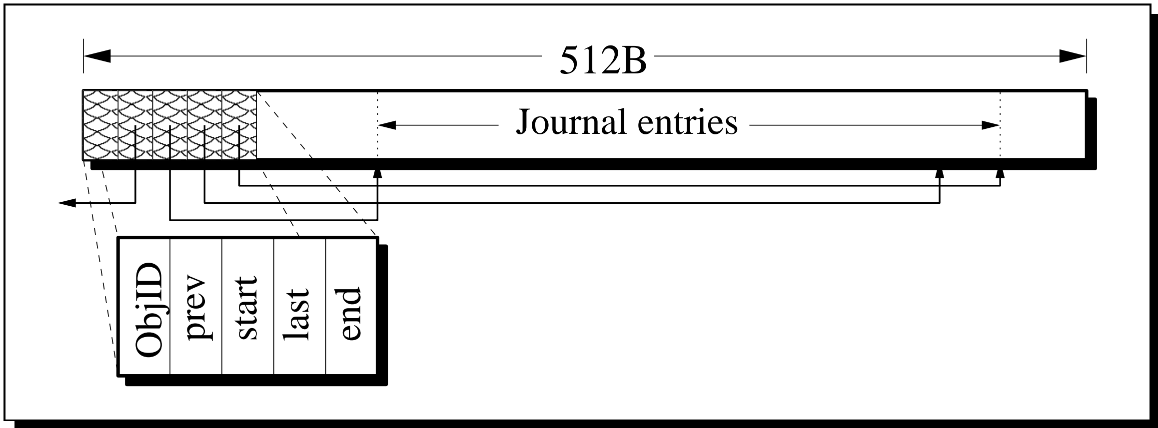

Figure 4: **Journal sector.** Contains packed journal entries.

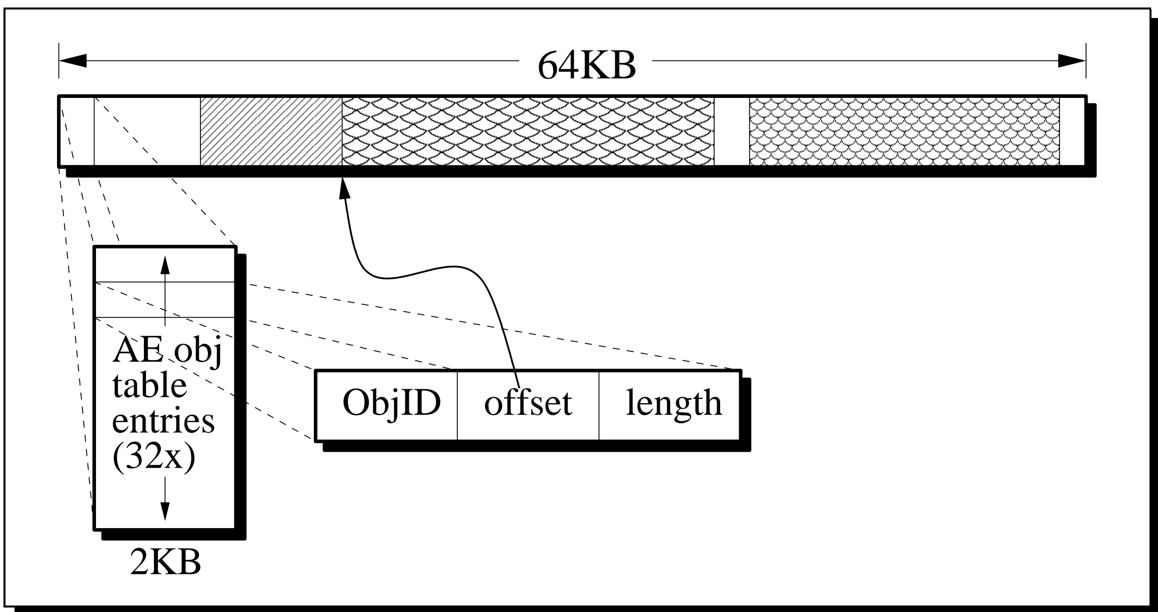

Figure 5: **Anti-entropy segment.** Holds the anti-entropy caches for various objects.

4.2.5 Cleaning

Like any log-structured file system, the S4 object system must use a cleaner to recover disk space. Unlike others, S4 can only reclaim sufficiently old versions. The cleaner runs during idle time, when specifically requested by an administrator, or when the system becomes low on disk space.

To reclaim disk space, the cleaner searches through the object map, looking for objects with an oldest time greater than the detection window. Once it finds an object, the cleaner searches back through the object's journal entries for resources to free. When it finds a part of

the object that is no longer within the history pool's window, it frees that space permanently. Then it takes note of other objects with data in that segment. Since all data in that segment was written within a small time period, the potential for possible cleaning is much higher on those objects. Concentrating the cleaner's efforts on specific groups of objects helps to increase the locality of the space freed and the likelihood of fully cleaning a segment. This is very important in a system such as S4, since old data cannot be moved without updating all of the associated metadata. Also, like LFS's cleaner, S4 suffers significant cleaning overhead when the drive gets too full.

4.3 S4 client module

Since the focus of self-securing storage is to provide an enhanced level of convenience and security on existing systems, we want to minimize changes to client systems. In keeping with this philosophy, the S4 drive is network-attached and an S4 client application serves as a user-level file system translator to avoid changes to the OS of client machines. The client translates requests from a file system on the target OS to S4 specific requests for objects. By running as a user-level process, without operating system modifications, the S4 client should port to different systems more easily.

The S4 client module currently has the capability to translate NFS version 2 requests to S4 requests. The S4 client appears to the local workstation as a NFS server. This simulated NFS server is mounted via the loopback interface to allow only that workstation access to the S4 client. The client receives the NFS requests and translates them into operations on S4 objects.

The implementation of the NFS file system on top of the S4 object store uses two types of objects: directory and file. Directory objects contain a list of ASCII filenames and their associated NFS file handles. File objects contain raw file data (or a symlink path). The NFS attributes structure is maintained within the opaque attribute space of each object.

When the S4 client receives an NFS request, the NFS file handle is directly hashed into the `ObjectID` of the directory or file. The S4 client can then make requests directly to the drive for the desired data.

In order to support the NFSv2 semantics, at the end of each NFS operation that modifies

the state of one or more objects, the client sends an additional RPC to the drive to flush buffered writes to the disk. Since this RPC does not return until the synchronization is complete, NFSv2 semantics are supported even though the drive normally buffers writes.

Because the client must overlay a file system on top of the flat object namespace, some operations require several drive operations (and hence several RPC calls) to implement a single NFS operation. These multiple operations are analogous to the multiple operations file systems must perform on block-based devices. To minimize the number of RPC calls necessary, the client aggressively maintains an attribute and a directory cache. The drive also supports combining the `SetAttr`, `GetAttr`, and `Sync0` operations with the `Create`, `Read`, `Write`, and `Append` operations.

5 Evaluation

This section evaluates the feasibility of self-securing storage, finding that it is possible for storage devices to assist with intrusion survival. Experiments with S4 indicate that device-controlled versioning can be done without significant performance reduction. Also, estimates of capacity growth based on reported workload characterizations indicate that history windows of a week or more can easily be supported in several real environments. It is important to note that although the prototype has been implemented on a Linux system, the drive should be viewed as a single-purpose embedded device.

5.1 Performance

Our main performance goal for S4 is to be comparable to other networked file systems. To explore this, we ran a number of micro and macro-benchmarks against a S4 drive mounted through the client NFS module. We compared these results to the FreeBSD 4.0 NFSv2 server. (Since the S4 drive runs on Linux, we would have preferred to to compare S4 against the Linux NFS server. However, Linux NFS does not comply with the NFSv2 semantics of committing data to stable storage before operation completion.)

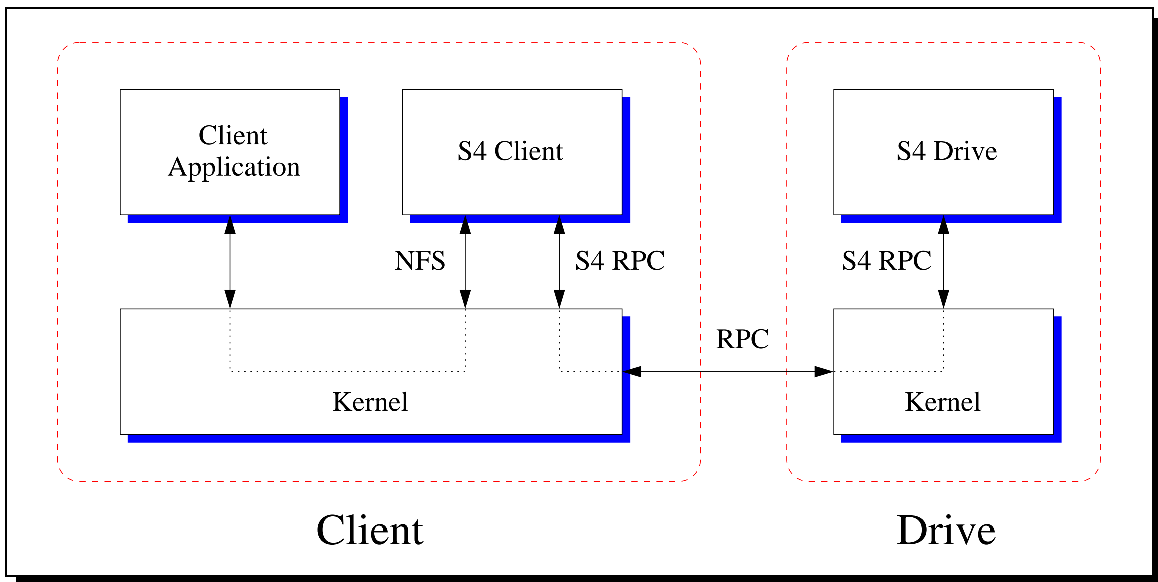

Figure 6: **S4 Client-Drive Configuration.**

5.1.1 Experimental setup

All experiments were run on three configurations: (1) a S4 drive running on RedHat 6.2 Linux communicating to a Linux client over S4 RPC through the S4 client module mounted via loopback (see Figure 6), (2) a BSD server communicating with a Linux client over UDP NFS, and (3) a BSD server communicating with a BSD client over TCP NFS (the BSD client is used because performance is much worse for a Linux client). In all cases, NFS was configured to use 4KB read/write transfer sizes, which is the only option supported by Linux. Read-ahead was disabled for TCP NFS and is not currently supported by the S4 client. Both BSD NFS configurations export a BSD FFS file system. All experiments were run a minimum of 5 times and have a standard deviation of less than 3% of the mean (unless reported otherwise). The S4 drives were configured with a 128MB buffer cache and an object cache capable of storing 8192 objects. These numbers were chosen given that the NFS servers' caches could grow to fill local memory (512MB). S4 was also configured with the anti-entropy cache and cleaner disabled.

In all experiments, the client system has a 550 MHz Pentium III, 128MB RAM, and a 3Com 3C905B 100Mb network adapter. The servers (or S4 drives) have of a 600 MHz Pentium III, 512MB RAM, a 9GB 10,000 RPM Ultra2 SCSI Seagate Cheetah drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel Etherexpress Pro100 100Mb net-

work adapter. The client and server are on the same subnet and are connected by a 100Mb network switch. All versions of Linux used an unmodified 2.2.14 kernel, and all BSD systems used a stock FreeBSD 4.0 installation.

5.1.2 Micro-benchmark results

To measure the performance of read, write, create, delete we ran benchmarks similar to those used by Rosenblum and Ousterhout to evaluate LFS [28]. In addition to creates, deletes, reads, and writes, these benchmarks also heavily test attribute and access control list operations, because NFS relies upon them heavily. The first benchmark measures small file performance; it consists of three phases: creation of 10,000 1KB files (split across 10 directories), reads of the newly created files in creation order, and deletion of the files in creation order. The second benchmark measures large file performance; it consists of four phases: sequential write, sequential read, random write, and random read. This benchmark either reads or writes a 100MB file in 4KB blocks. For each of these benchmarks, the cache was flushed between each phase.

Figure 7 shows the results of the small file benchmark. Overall, S4/NFS performance is similar to BSD/NFS. S4 performs slightly better on creates and deletes, due to a decreased number of disk I/Os. Since S4 is log-structured, only one disk write is needed to create the file and add the directory entry; FFS needs at least two. (Recall that delayed writes cannot be used without NVRAM, because NFS semantics require immediate persistence.) Reads are slower than the BSD UDP and TCP configurations, due to an artifact of S4 client directory management.

Figure 8 shows the results of the large file benchmark. Again, S4/NFS performance is similar to BSD/NFS. S4 performs somewhat better in all cases except for the read sequential case. Random and sequential writes are better due to the log structure of the drive. Random reads are significantly better due to S4 reading in entire segments at a time (64KB), and due to the fact that the entire file can fit in the cache (as it can in the case of NFS). (Sequential writes in BSD/UDP NFS had a standard deviation of 10%.)

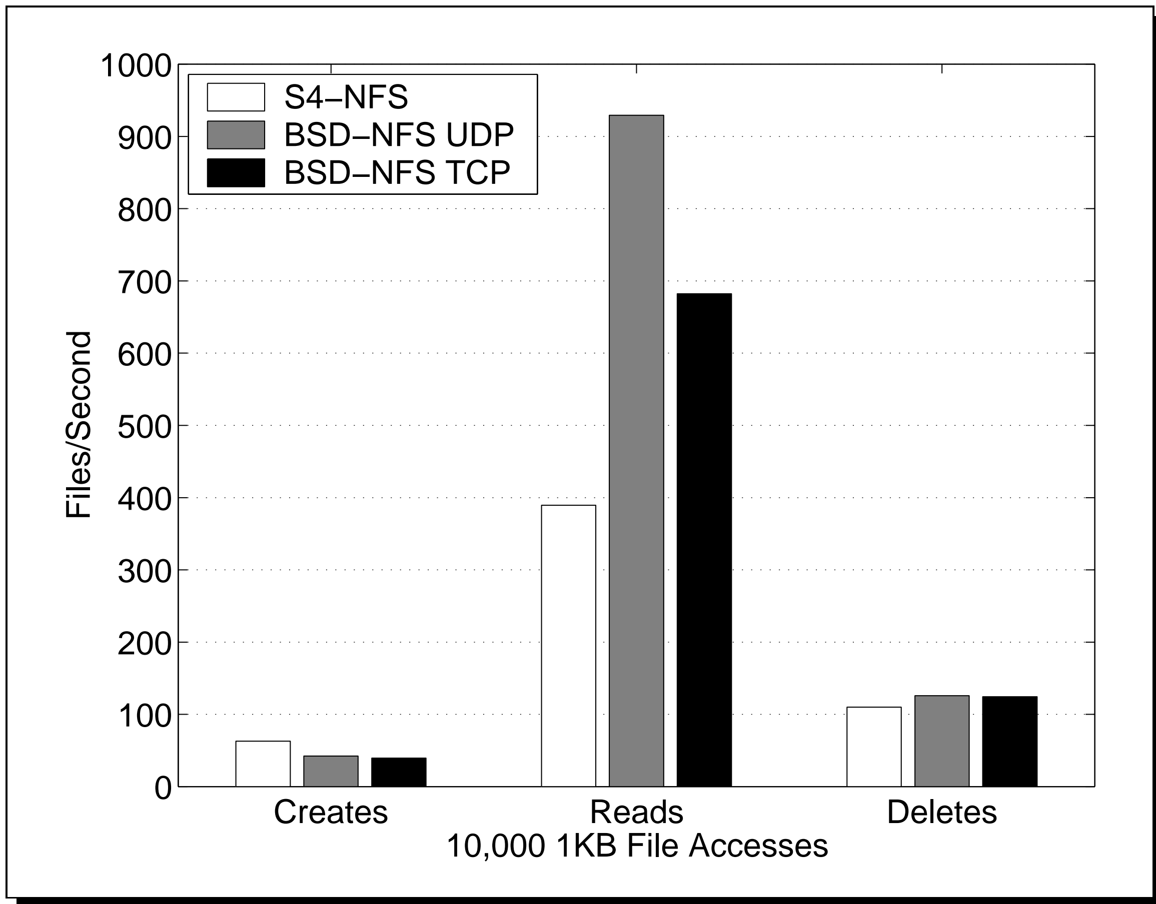

Figure 7: **Small-file performance.**

5.1.3 Macro-benchmark results

To evaluate performance for more realistic workloads, we present results from two macro-benchmarks: the Postmark benchmark [14] and the SSH-build benchmark [37]. These benchmarks crudely represent Internet server and software development workloads.

`Postmark` was designed to measure the performance of a file system used for electronic mail, netnews, and web based services. It does this by creating a large number of small randomly-sized files (between 512B and 9KB for our tests), on which a specified number of transactions are performed. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The transaction types are chosen randomly with consideration given to user definable weights. Our configuration consists of 20,000 transactions on 5,000 files, with a file size of between 512B and 9KB. The biases for transaction type are set equal.

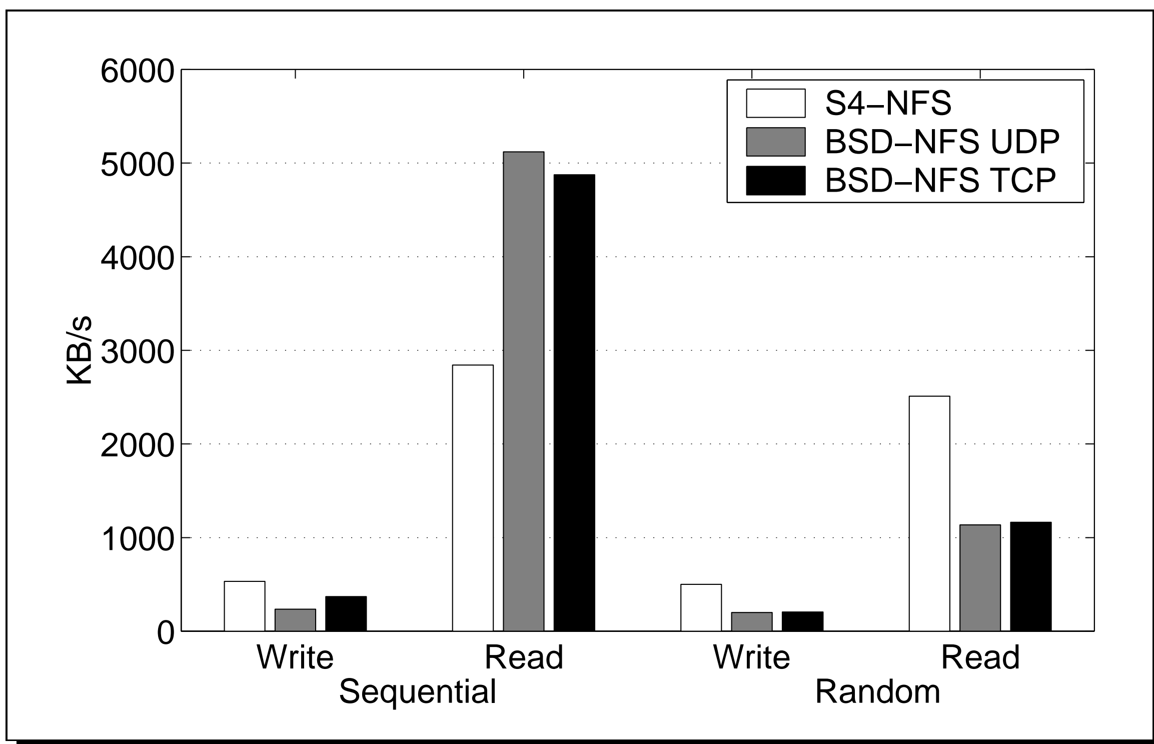

Figure 8: **Large-file performance.**

The results of the postmark benchmark are shown in Figure 9. Each bar shows the total running time in seconds split between initial file creation time (creating initial 5000 files) and the time to complete all transactions. S4/NFS outperforms the other configurations because of its superior performance for non-sequential small file operations.

The `SSH-build` benchmark was constructed as a replacement for the Andrew file system benchmark [12]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27. (SSH is approximately 1MB in size before decompression) This phase stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables.

The times of `SSH-build's` three phases are shown in Figure 10. Performance is similar across the 3 configurations. S4 outperforms the BSD configurations in all phases due to the relative performance advantages of creates, deletes, and writes.

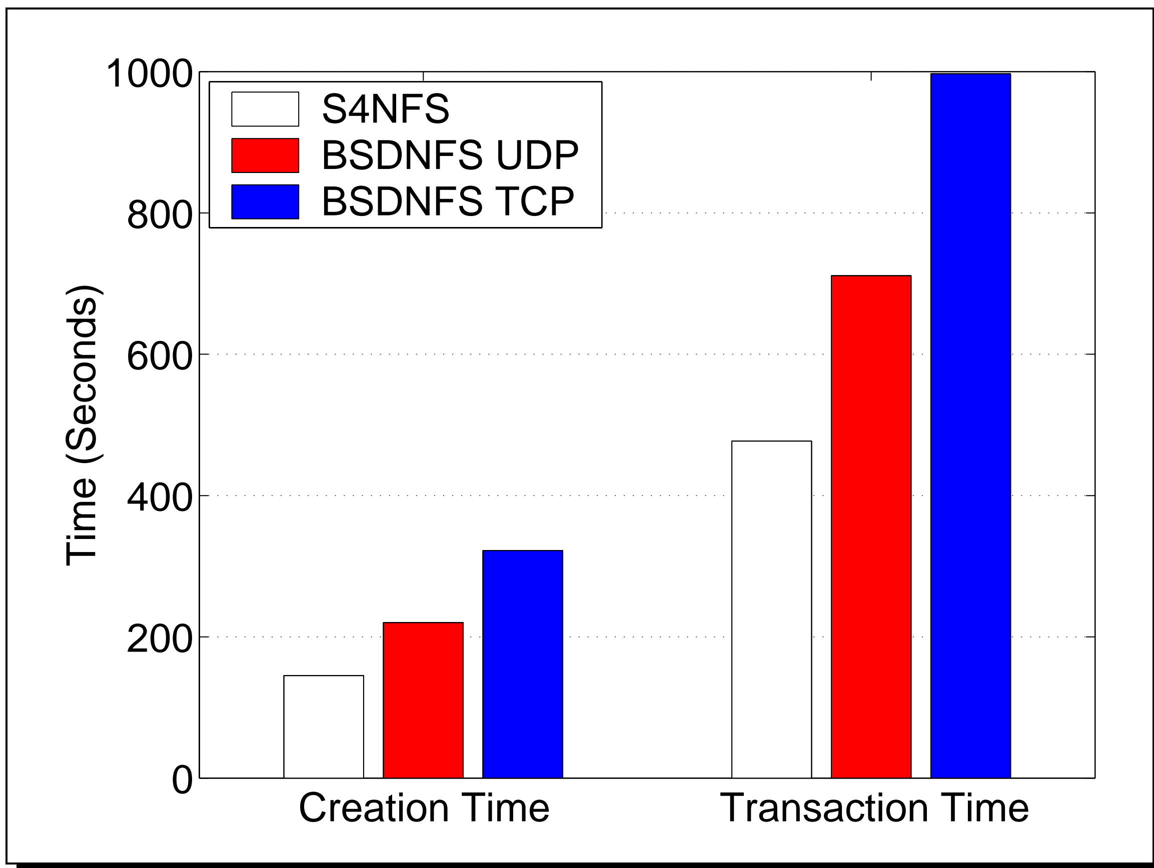

Figure 9: **Postmark benchmark.**

Year	Environment	Ref	System	Data rate	Detection window
1996	Carnegie Mellon	[32]	AFS	143MB per day	71 days
1999	Cornell	[35]	Windows NT	1GB per day	10 days
1999	HP Research labs	[29]	HP-UX	108MB per day	94 days

Table 2: **Space usage survey.** This table shows the expected detection window that could be provided by utilizing 10GB of a modern disk drive, assuming no differencing or compression. This conservative history pool would consume only 20% of a 50GB disk's total capacity.

5.2 Capacity requirements

To evaluate the capacity required to maintain a week-long history pool, we examine data from three recent workload studies (see Table 2). The AFS trace study [32] reports 143MB per day of write traffic. Even if we pessimistically increase this number to 1GB per day, as was observed in Vogels' Windows NT file usage study [35], it is not unreasonable to believe that seven days worth of data could be kept on a modern 50GB disk. The Elephant paper [29]

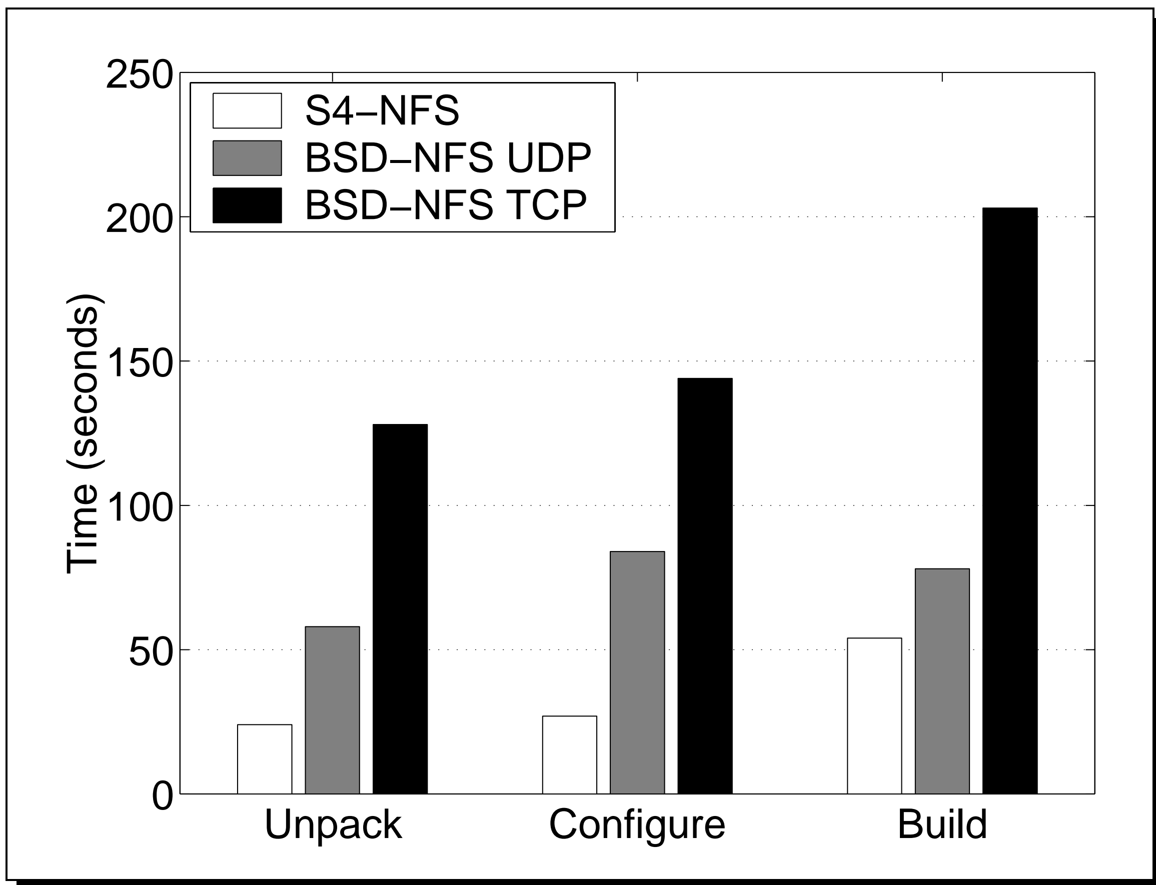

Figure 10: **SSH - Unpack, configure, and build.**

presents a data rate of 110MB written per day, again over a week of changes could easily be kept on a modern disk.

Much work has been done in evaluating the efficiency of differencing and compression [2, 3, 4]. To briefly explore the potential benefits for S4, we retrieved the code base for S4 from the CVS repository at a single point each day for a week. After compiling the code, both differencing and combined differencing + compression were applied between each tree and its direct neighbor in time using xdelta [19, 20]. The total sizes of all seven trees, the object files of all the trees, and the source files of all the trees were then compared to their respective differenced sizes. For all versions of all files, 250MB of storage is needed. This number drops to 80MB with differencing and 50MB with differencing + compression.

6 Discussion

This section discusses several important implications of self-securing storage.

Selective versioning: There are data that users would prefer not to have backed up at all. The common approach to this is to store them in directories known to be skipped by the backup system. Since one of the goals of S4 is to allow recovery of exploit tools, it does not support designating objects as non-versioned. A system may be configured with non-S4 partitions to support selective versioning. While this would provide a way to prevent versioning of temporary files and other non-critical data, it would also create a location where an intruder could store exploit tools without fear that they will be recovered.

Versioning file systems vs. self-securing storage: Versioning file systems excel at providing users with a safety net for recovery from accidents. They can maintain old file versions long after they would be reclaimed by the S4 system, but provide little additional system security. This is because they rely on the host's OS for security and aggressively prune apparently insignificant versions. By combining self-securing storage with a versioning file system, recovery from users' accidents could be well-supported while also maintaining the benefits of intrusion survival.

Client-side cache effects: In order to improve efficiency, most client systems use caches minimize storage latencies. This is at odds with the desire to to have the device audit users' accesses and capture exploit tools. Client-side read caches hide data dependency information that would otherwise be available to the drive in the form of reads followed quickly by writes. However, this information could be provided by client systems as (questionable) hints during writes. Write caches cause a more serious problem when files are created then quickly deleted, thus never being sent to the drive. This could cause difficulties capturing exploit tools since they may never be written to the drive. While the client cache effects may obscure some of the activity in the system, users' data that is stored on the device is still completely protected.

Object-based vs. block-based storage: Attempting to implement a self-securing storage device with a block device presents several problems. Since objects are designed to contain one data item (file or directory), enforcing access control at this level is more

manageable than attempting to properly assign permissions on a per-block basis. In addition, maintaining versions of objects as a whole, rather than having to collect and correlate individual blocks, simplifies recovery tools and internal reorganization mechanisms, like the anti-entropy cache. Still, although some of S4's benefits would be lost, we see no roadblock to self-securing block-based storage.

Multi-device coordination: Multi-device coordination is necessary for operations such as striping data or implementing RAID on multiple disks. In addition to the normal coordination that is necessary to make sure that the multiple copies of data are synchronized, recovery operations must also coordinate old versions to make sure that the objects are consistent when they are recovered from the history pool. On the other hand, clusters of self-securing storage devices could maintain a single history pool and balance the load of versioning objects.

7 Related Work

Self-securing storage and S4 build on many ideas from previous work. Perhaps the clearest example is versioning: many versioned file systems have helped their users to recover from mistakes [22, 10]. Santry, et. al, provides a good discussion of techniques for traversing versions and deciding what to retain [29]. S4's history pool corresponds to Elephant's "keep all" policy (during its time window), and it uses Elephant's time-based access. The largest advantage of S4 over previous versioning systems is that it has been partitioned from the operating system. While this creates another level of indirection, it adds to the survivability of the storage.

S4's device-embedded storage management is another instance of many recent "smart disk" systems [1, 7, 15, 27, 36]. All of these exploit the increasing computation power of such devices. Some also put these devices on networks and exploit an object-based interface. There is now an ANSI X3T10 (SCSI) working group looking to create a new standard for object-based storage devices. The S4 interface is similar to these.

The standard method of intrustion recovery is to keep a periodic backup of files on trusted storage. Several file systems simplify and extend this process by allowing a snapshot

to be taken of a file system [11, 12, 18]. This snapshot can then be accessed through the standard file system tools for file retrieval. Spiralog [13] uses a log-structured file system to allow for backups to be made during system operation by simply recording the entire log to tertiary storage. While these systems are effective in preventing the loss of existing critical data, the window of time in which data can be destroyed or tampered with is much larger than S4, often up to 24 hours, and is reliant upon a system administrator for operation. Also, intrusion diagnosis is extremely difficult in such systems. Permanent file storage [26] provides an unlimited set of intruder-proof backups over time. These systems are unlikely to become the first line of storage because of lengthy access times.

S4 borrows on-disk data structures from several systems. Unlike Elephant's FFS-like layout [23], the disk layout of S4 more closely resembles that of a log structured file system [28]. Log-structuring is designed to optimize write performance at a potential cost to read performance. S4's anti-entropy cache is reminiscent of several proposals for background reorganization for improving read performance [21, 30]. Many file systems use journaling to improve performance while maintaining disk consistency [5, 31, 34], deleting the journal information once checkpoints ensure that the corresponding blocks are all on disk. S4 uses journaling to persistently maintain metadata versions in a space-efficient manner.

8 Conclusions

Self-securing storage ensures data and audit log survival in the presence of successful intrusions and even compromised host operating systems. Experiments with the S4 prototype show that self-securing storage devices can achieve performance that is comparable to standard storage appliances. In addition, analysis of recent workload studies suggest that complete version histories can be kept for several weeks on state-of-the-art disk drives.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, California), pages 81–91. ACM, 3–7 October 1998.

[2] Randal C. Burns. *Differential compression: a generalized solution for binary files.* Technical report. December 2000.

[3] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm.* 124. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 10 May 1994.

[4] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):2–9, October 1992.

[5] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. *Annual USENIX Technical Conference* (San Francisco, CA), pages 43–60, Winter 1992.

[6] Dorothy Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, **SE-13**(2):222–232, February 1987.

[7] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[8] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. NASD scalable storage systems. *USENIX.99* (Monterey, CA., June 1999), 1999.

[9] Howard Gobioff. *Security for a high performance commodity storage subsystem*. PhD thesis, published as TR CMU–CS–99–160. Carnegie-Mellon University, Pittsburgh, PA, July 1999.

[10] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. *ACM Symposium on Operating System Principles* (Austin, Texas). Published as *Operating Systems Review*, **21**(5):155–162, November 1987.

[11] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA). Published as *Proceedings of USENIX*, pages 235–246. USENIX Association, 19 January 1994.

[12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[13] James E. Johnson and William A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, **8**(2):5–14, 1996.

[14] Jeffrey Katcher. *PostMark: a new file system benchmark*. TR3022. Network Apliance, October 1997.

[15] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, **27**(3):42–52, September 1998.

[16] Stephen Kent and Randall Atkinson. *Security Architecture for the Internet Protocol*, RFC–2401, November 1998.

[17] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. *Conference on Computer and Communications Security* (Fairfax, Virginia), pages 18–29, 2–4 November 1994.

[18] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA). Published as *SIGPLAN Notices*, **31**(9):84–92, 1–5 October 1996.

[19] Josh MacDonald. Verioned File Archiving, Compression, and Distribution, 2000.

[20] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. *European Conference on Object-Oriented Programming* (Brussels, Belgium, July, 20–21). Published as *Proceedings of ECOOP*, pages 33–45. Springer-Verlag, 1998.

[21] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.

[22] K. McCoy. *VMS file system internals*. Digital Press, 1990.

[23] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[24] Sun Microsystems. *RPC: remote procedure call protocol specification version 2*, RFC–1057, June 1988.

[25] *Object based storage devices: a command set proposal*. Technical report. October 1999. http://www.T10.org/.

[26] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *UKUUG Summer* (London), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

[27] Erik Riedel and Garth Gibson. *Active disks-remote execution for network-attached storage*. TR CMU-CS-97-198. December 1997.

[28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[29] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Ross W. Carton, Jacob Ofir, and Alistair C. Veitch. Deciding when to forget in the Elephant file system. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, South Carolina). Published as *Operating Systems Review*, **33**(5):110–123. ACM, 12–15 December 1999.

[30] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Annual USENIX Technical Conference* (New Orleans), pages 249–264. Usenix Association, 16–20 January 1995.

[31] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA), 18–23 June 2000.

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

[33] Sun Microsystems Incorporated, 2550 Garcia Ave, Mountain View, CA 94043. *Remote Procedure Call Protocol Specification*, Part number 800–1177–01, revision A–β, January 1985.

[34] Adam Sweeney. Scalability in the XFS file system. *USENIX*. (San Diego, California), pages 1–14, 22–26 January 1996.

[35] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.

[36] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, Winter 1998.

[37] Tatu Ylonen. SSH - Secure login connections over the internet. *USENIX Security Symposium* (San Jose, CA). USENIX Association, 22–25 July 1996.