Appears in Proceedings of the 4th Symposium on Operating Systems Design and Implementation.
San Diego, CA. October, 2000.

Self-Securing Storage: Protecting Data in Compromised Systems

John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules
Gregory R. Ganger
Carnegie Mellon University

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 old versions of data for a window of time, regardless of the commands received from potentially compromised host operating systems. Within the window, system administrators have this valuable information for intrusion diagnosis and recovery. Our implementation, called S4, combines log-structuring with journal-based metadata to minimize the performance costs of comprehensive versioning. Experiments show that self-securing storage devices can deliver performance that is comparable with conventional storage systems. In addition, 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.

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 still 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 backup (i.e., one created prior to the intrusion), reinitializing the system, and restoring information from the backup. 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 backup 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 OSes, self-securing storage devices view them, and their users, as questionable entities for which they work. These self-contained, self-controlled devices internally version all data and audit all requests for a guaranteed amount of time (e.g., a week or a month), thus providing system administrators time to detect intrusions. For intrusions detected within this window, all of the version and audit information is available for analysis and recovery. 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 complex OSes or their 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 server, 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, storage densities increase faster than most computer characteristics (100%+ per annum in recent years). Analysis of recent workload studies [29, 34] suggests that it is possible to version all data on modern 30–100GB drives for several weeks. Further, aggressive compression and cross-version differencing techniques can extend the intrusion detection window offered by self-securing storage devices. Other challenges include efficiently encoding the many metadata changes, achieving secure administrative control, and dealing with denial-of-service attacks.

The S4 system addresses these challenges with a new storage management structure. Specifically, S4 uses a log-structured object system for data versions and a novel journal-based structure for metadata versions. In addition to reducing space utilization, journal-based metadata 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 S4-enhanced NFS is comparable to FreeBSD's NFS for both micro-benchmarks and application benchmarks. The fundamental costs associated with self-securing storage degrade performance by less than 13% relative to similar systems that provide no data protection guarantees.

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, identifies 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 performance and capacity overheads of self-securing storage. Section 6 discusses a number of 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 access at will in the future. They may also install other software, read and modify sensitive files, or use the system as a platform for launching additional attacks. Depending on the skill with which the intruders hide their 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 intruders can continue their malicious activities while users continue to use the system, thus entangling legitimate changes with those of the intruders. 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 "administrator" 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 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 them.

2.1 Diagnosis

Intrusion diagnosis consists of three phases: detecting the intrusion, discovering what weaknesses were exploited (for future prevention), and determining what the intruder did. 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 alert users and system administrators noticing odd behavior. Examining the system logs is the most common approach to intrusion detection [7], 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 [16]. Such systems are vulnerable to intruders that can change what the IDS thinks is a "safe" copy.

Determining how an intruder compromised the system is often impossible in conventional systems, because he will scrub the system logs. In addition, any *exploit tools* (utilities for compromising computer systems) that may have been stored on the target machine for use in multi-stage intrusions are usually 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 in diagnosing an intrusion is to discover what was accessed and modified by the intruder. This is 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.

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 data, modified since the intrusion, be saved. 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 modified data is very difficult, and overlooked tampering 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 pre-intrusion backup, the legitimately modified 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 host operating systems, relying on them to protect users' data. A self-securing storage device operates as an independent entity, tasked with the responsibility of not only storing data, but protecting it. 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 previous versions of the 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 this type of device.

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 and version information also helps to diagnose intrusions and detect the propagation of maliciously modified data.

Self-securing storage simplifies detection of an intrusion since versioned system logs cannot be imperceptibly altered. In addition, modified system executables are easily noticed. Because of this, self-securing storage makes conventional tamper detection systems obsolete.

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 doctored, 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 can 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 backup or up-to-date checksums (for tamper detection) of system files. After such restoration, critical data can be incrementally recovered from the history pool. Additionally, by utilizing the storage device's audit log, it is possible to assess which data might have been directly affected by the intruder.

The data protection that self-securing storage provides 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 exports 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 singlefunction device, the task of making it secure is much easier; compromising its firmware is analogous to breaking into an IDE or SCSI disk.

The actual protocol used to communicate with the storage device does not affect the data integrity that the new security perimeter provides. The choice of protocol does, however, affect the usefulness of the audit log in terms of the actions it can record and its correctness. For instance, the NFS protocol provides no authentication or integrity guarantees, therefore the audit log may not be able to accurately link a request with its originating client. Nonetheless, the principles of self-securing storage apply equally to "enhanced" disk drives, network-attached storage servers, and file servers.

For network-attached storage devices (as opposed to devices attached directly to a single host system), the new security perimeter becomes more useful if the device can verify each access as coming from both a valid user and a valid client. Such verification allows the device to enforce access control decisions and partially track propagation of tainted data. If clients and users are authenticated, accesses 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 or user account.

3.3 History pool management

The old versions of objects kept by the device 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 an old version will age and have its space reclaimed. Because clients cannot be trusted to demarcate versions consisting of multiple modifications, a separate version should 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 guaranteed 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.

Although the capacity of disk drives is growing at a remarkable rate, it is still finite, which poses 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 conventional 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. 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; although this will allow recovery of 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 flawed 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. 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 use 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 a safe algorithm that would save space in the history pool by pruning versions within the detection window. Almost any algorithm that selectively removes 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 History pool access control

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, but 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. Although 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.

Our compromise is to allow users to selectively make this decision. By choice, a user could thus delete an object, version, or all versions from visibility by anyone other than the administrator, since permanent deletion of data via any other method than aging would be unsafe. This choice allows 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 Administrative access

A method for secure administrative access is needed for the necessary but dangerous commands that a self-securing storage device must support. Such commands include setting the guaranteed detection window, erasing parts of the history pool, and accessing data that users have marked as "unrecoverable." Such administrative access can be securely granted in a number of ways, including physical access (e.g., flipping a switch on the device) or well-protected cryptographic keys.

Administrative access is not necessary for users attempting to recover their own files from accidents. Users' accesses to the history pool should be handled with the same form of protection used for their normal accesses. This is acceptable for user activity, since all actions permitted for ordinary users can be audited and repaired.

3.6 Version and 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 versions that are stored by 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`. This is accomplished by extending the read interfaces of the device to include an optional time parameter. When this parameter is specified, the drive returns data from the version of the object that was valid at the requested time.

In addition to providing 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 was written. Such a link between a source file and its corresponding object file 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 S4 Implementation

S4 is a self-securing storage server that transparently maintains an efficient object-versioning system 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.

RPC Type	**Allows Time-Based Access**	**Description**
`Create`	no	Create an object
`Delete`	no	Delete an object
`Read`	yes	Read data from an object
`Write`	no	Write data to an object
`Append`	no	Append data to the end of an object
`Truncate`	no	Truncate an object to a specified length
`GetAttr`	yes	Get the attributes of an object (S4-specific and opaque)
`SetAttr`	no	Set the opaque attributes of an object
`GetACLByUser`	yes	Get an ACL entry for an object given a specific `UserID`
`GetACLByIndex`	yes	Get an ACL entry for an object by its index in the object's ACL table
`SetACL`	no	Set an ACL entry for an object
`PCreate`	no	Create a partition (associate a name with an `ObjectID`)
`PDelete`	no	Delete a partition (remove a name/`ObjectID` association)
`PList`	yes	List the partitions
`PMount`	yes	Retrieve the `ObjectID` given its name
`Sync`	not applicable	Sync the entire cache to disk
`Flush`	not applicable	Removes all versions of all objects between two times
`FlushO`	not applicable	Removes all versions of an object between two times
`SetWindow`	not applicable	Adjusts the guaranteed detection window of the S4 device

Table 1: **S4 Remote Procedure Call List** – Operations that support time-based access accept a *time* in addition to the normal parameters; this *time* is used to find the appropriate version in the history pool. Note that all modifications create new versions without affecting the previous versions.

4.1 A self-securing object store

S4 is a network-attached object store with an interface similar to recent object-based disk proposals [9, 24]. This interface simplifies access control and internal performance enhancement relative to a standard block interface.

In S4, objects exist in a flat namespace managed by the "drive" (i.e., the object store). 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 that specifies which entities (users and client machines) have permission to access the object. Objects also have metadata, file data, and opaque attribute space (for use by client file systems) associated with them.

To enable persistent mount points, a S4 drive supports "named objects." The object names are an association of an arbitrary ASCII string with a particular `ObjectID`. The table of named objects is implemented as a special S4 object accessed through dedicated partition manipulation RPC calls. This table is versioned in the same manner as other objects on the S4 drive.

4.1.1 S4 RPC interface

Table 1 lists the RPC commands supported by the S4 drive. The read-only commands (`read`, `getattr`, `getacl`, `plist`, and `pmount`) accept an optional *time* parameter. When the *time* is provided, the drive performs the read request on the version of the object that was "most current" at the time specified, provided that the user making the request has sufficient privileges.

The ACLs associated with objects have the traditional set of flags, with one addition—the `Recovery` flag. The `Recovery` flag determines whether or not a given user may read (recover) an object version from the history pool once it is overwritten or deleted. When this flag is clear, only the device administrator may read this object version once it is pushed into the history pool. The `Recovery` flag allows users to decide the sensitivity of old versions on a file-by-file basis.

4.1.2 S4/NFS translation

Since one goal of self-securing storage is to provide an enhanced level of security and convenience on existing systems, the prototype minimizes changes to client systems. In keeping with this philosophy, the S4 drive is network-attached and an "S4 client" daemon serves as a user-level file system translator (Figure 1a). The S4 client translates requests from a file system on the target OS to S4-specific requests for objects. Because it runs as a user-level process, without operating system modifications, the S4 client should port to different systems easily.

The S4 client currently has the ability to translate NFS version 2 requests to S4 requests. The S4 client appears to the local workstation as a NFS server. This emulated 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 S4 operations. NFSv2 was chosen over version 3 because its client is well-supported within Linux, and its lack of write caching allows the drive to maintain a detailed account of client actions.

Figure 1 shows two approaches to using the S4 client to serve NFS requests with the S4 drive. The first places the S4 client on the client system, as described previously, and uses the S4 drive as a network-attached storage device. The second incorporates the S4 client functionality into the server, as a NFS-to-S4 translator. This configuration acts as a S4-enhanced NFS server (Figure 1b) for normal file system activity, but recovery must still be accomplished through the S4 protocol since the NFS protocol has no notion of "time-based" access.

The implementation of the NFS file system overlays files and directories on top of S4 objects. Objects used as directories contain a list of ASCII filenames and their associated NFS file handles. Objects used as files and symlinks contain the corresponding data. The NFS attribute structure is maintained within the opaque attribute space of each object.

When the S4 client receives a NFS request, the NFS file handle (previously constructed by the S4 client) can be 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.

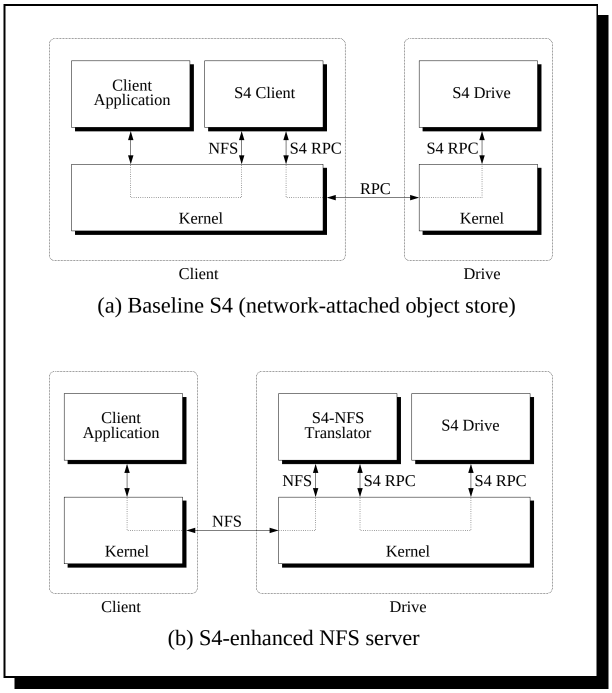

Figure 1: **Two S4 Configurations** – This figure shows two S4 configurations that provide self-securing storage via a NFS interface. (a) shows S4 as a network-attached object store with the S4 client daemon translating NFS requests to S4-specific RPCs. (b) shows a self-securing NFS server created by combining the NFS-to-S4 translation and the S4 drive.

To support NFSv2 semantics, the client sends an additional RPC to the drive to flush buffered writes to the disk at the end of each NFS operation that modifies the state of one or more objects. Since this RPC does not return until the synchronization is complete, NFSv2 semantics are supported even though the drive normally caches writes.

Because the client overlays a file system on top of the flat object namespace, some file system operations require several drive operations (and hence RPC calls). These sets of operations are analogous to the operations that file systems must perform on block-based devices. To minimize the number of RPC calls necessary, the S4 client aggressively maintains attribute and directory caches (for reads only). The drive also supports batching of `setattr`, `getattr`, and `sync` operations with `create`, `read`, `write`, and `append` operations.

4.2 S4 drive internals

The main goals for the S4 drive implementation 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 requires a combination of known and novel techniques for organizing on-disk data.

4.2.1 Log-structuring for efficient writes

Since data within the history pool cannot be overwritten, the S4 drive uses a log structure similar to LFS [27]. This structure allows multiple data and metadata updates to be clustered into fewer, larger writes. Importantly, it also obviates any need to move previous versions before writing.

In order to prune old versions and reclaim unused segments, S4 includes a background cleaner. While the goal of this cleaner is similar to that of the LFS cleaner, the design must be slightly different. Specifically, deprecated objects cannot be reclaimed unless they have also aged out of the history pool. Therefore, the S4 cleaner searches through the object map for objects with an oldest time greater than the detection window. Once a suitable object is found, the cleaner permanently frees all data and metadata older than the window. If this clears all of the resources within a segment, the segment can be marked as free and used as a fresh segment for foreground activity.

4.2.2 Journal-based metadata

To efficiently keep all versions of object metadata, S4 uses *journal-based metadata*, which replaces most instances of metadata with compact journal entries.

Because clients are not trusted to notify S4 when objects are closed, every update creates a new version and thus new metadata. For example, when data pointed to by indirect blocks is modified, the indirect blocks must be versioned as well. In a conventional versioning system, a single update to a triple-indirect block could require four new blocks as well as a new inode. Early experiments with this type of versioning system showed that modifying 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.

In order to significantly reduce these problems, S4 encodes metadata changes in a journal that is maintained for the duration of the detection window. By persistently keeping journal entries of all metadata changes, metadata writes can be safely delayed and coalesced, since individual inode and indirect block versions can be recreated from the journal. To avoid rebuilding an object's current state from the journal during normal operation, an object's metadata is checkpointed to a log segment before being evicted from the cache. Unlike conventional journaling, such checkpointing does not prune journal space; only aging may prune space. Figure 2 depicts the difference in disk space usage between journal-based metadata and conventional versioning when writing data to an indirect data block.

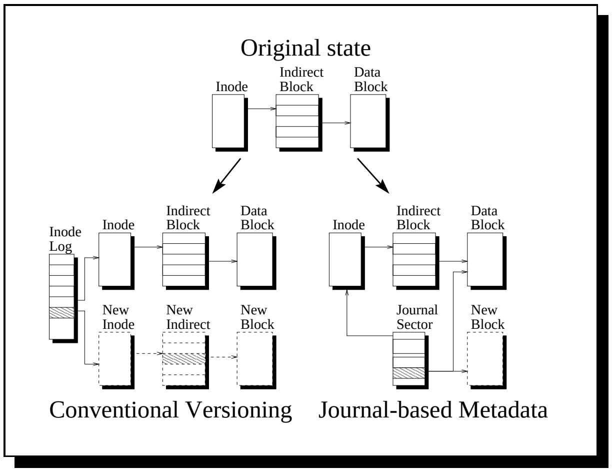

Figure 2: **Efficiency of Metadata Versioning** – The above figure compares metadata management in a conventional versioning system to S4's journal-based metadata approach. When writing to an indirect block, a conventional versioning system allocates a new data block, a new indirect block, and a new inode. Also, the identity of the new inode must be recorded (e.g., in an Elephant-like inode log). With journal-based metadata, a single journal entry suffices, pointing to both the new and old data blocks.

In addition to the entries needed to describe metadata changes, a `checkpoint` entry is needed. This `checkpoint` entry denotes writing a consistent copy of all of an object's metadata to disk. It is necessary to have at least one checkpoint of an object's metadata on disk at all times, since this is the starting point for all time-based and crash recovery recreations.

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 segment. The sectors are identified by segment summary information. Journal sectors are chained together backward in time to allow for version reconstruction.

Journal-based metadata can also simplify cross-version differential compression [3]. Since the blocks changed between versions are noted within each 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. Cross-version differencing of old data will often be effective in reducing the amount of space used by old versions. Adding differencing technology into the S4 cleaner is an area of future work.

4.2.3 Audit log

In addition to maintaining previous object versions, S4 maintains an append-only audit log of all requests. This log is implemented as a reserved object within the drive that cannot be modified except by the drive itself. However, it can be read via RPC operations. The data written to the audit log includes command arguments as well as the originating client and user. All RPC operations (read, write, and administrative) are logged. Since the audit log may only be written by the drive front end, it need not be versioned, thus increasing space efficiency and decreasing performance costs.

5 Evaluation of self-securing storage

This section evaluates the feasibility of self-securing storage. Experiments with S4 indicate that comprehensive versioning and auditing can be performed without a significant performance impact. Also, estimates of capacity growth, based on reported workload characterizations, indicate that history windows of several weeks can easily be supported in several real environments.

5.1 Performance

The main performance goal for S4 is to be comparable to other networked file systems while offering enhanced security features. This section demonstrates that this goal is achieved and also explores the overheads specifically associated with self-securing storage features.

5.1.1 Experimental Setup

The four systems used in the experiments had the following configurations: (1) a S4 drive running on RedHat 6.1 Linux communicating with a Linux client over S4 RPC through the S4 client module (Figure 1a), (2) a S4-enhanced NFS server running on RedHat 6.1 Linux communicating with a Linux client over NFS (Figure 1b), (3) a FreeBSD 4.0 server communicating with a Linux client over NFS, and (4) a RedHat 6.1 Linux server communicating with a Linux client over NFS. Since Linux NFS does not comply with the NFSv2 semantics of committing data to stable storage before operation completion, the Linux server's file system was mounted synchronously to approximate NFS semantics. In all cases, NFS was configured to use 4KB read/write transfer sizes, the only option supported by Linux. The FreeBSD NFS configuration exports a BSD FFS file system, while the Linux NFS configuration exports an ext2 file system. All experiments were run five times and have a standard deviation of less than 3% of the mean. The S4 drives were configured with a 128MB buffer cache and a 32MB object cache. The Linux and FreeBSD NFS servers' caches could grow to fill local memory (512MB).

In all experiments, the client system has a 550MHz Pentium III, 128MB RAM, and a 3Com 3C905B 100Mb network adapter. The servers have a 600MHz 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 network adapter. The client and server are on the same subnet and are connected by a 100Mb network switch. All versions of Linux use an unmodified 2.2.14 kernel, and the BSD system uses a stock FreeBSD 4.0 installation.

To evaluate performance for common workloads, results from two application benchmarks are presented: the PostMark benchmark [14] and the SSH-build benchmark [36]. These benchmarks crudely represent Internet server and software development workloads, respectively.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services. It creates a large number of small randomly-sized files (between 512B and 9KB) and performs a specified number of transactions on them. Each transaction consists of two subtransactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction type are equal.

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 (approximately 1MB in size before decom-

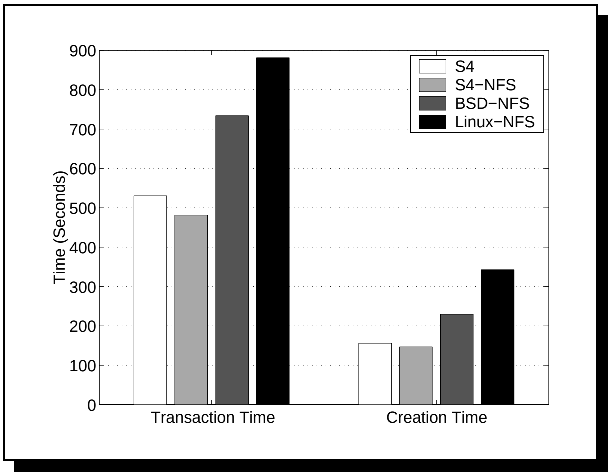

Figure 3: **PostMark Benchmark**

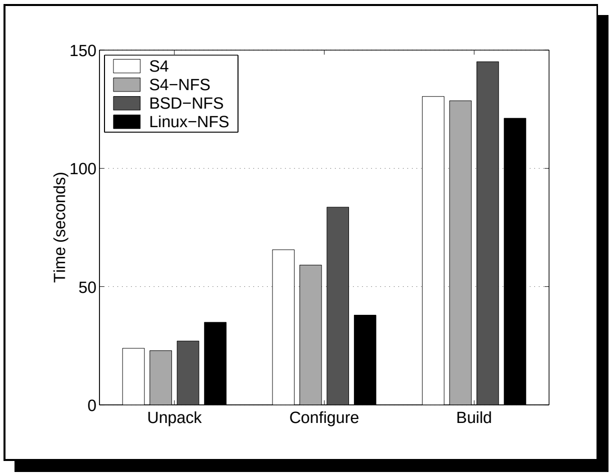

Figure 4: **SSH-build Benchmark**

pression), 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.

5.1.2 Comparison of the servers

To gauge the overall performance of S4, the four systems described earlier were compared. As hoped, S4 performs comparably to the existing NFS servers.

Figure 3 shows the results of the PostMark benchmark. The times for both the creation (time to create the initial 5000 files) and transaction phases of PostMark are shown for each system. The S4 systems' performance is similar to both BSD and Linux NFS performance, doing slightly better due to their log structured layout.

The times of `SSH-build's` three phases are shown in Figure 4. Performance is similar across the S4 and BSD configurations. The superior performance of the Linux NFS server in the configure stage is due to a much lower number of write I/Os than in the BSD and S4 servers, apparently due to a flaw in the synchronous mount option under Linux.

5.1.3 Overhead of the S4 cleaner

In addition to the more visible process of creating new versions, S4 must eventually garbage collect data that has expired from the history pool. This garbage collection comes at a cost. The potential overhead of the cleaner was measured by running the PostMark benchmark with 50,000 transactions on increasingly large sets of initial files. For each set of initial files, the benchmark was run once with the cleaner disabled and once with the cleaner competing with foreground activity.

The results shown in Figure 5 represent PostMark running with the initial set of files filling between 2% and 90% of a 2GB disk. As expected, when the working set increases, performance of the normal S4 system degrades due to increasingly poor cache and disk locality. The sharp drop in the graph from 2% to 10% is caused by the fact that the set of files and data expands beyond the bounds of the drive's cache.

Although the S4 cleaner is slightly different, it was expected to behave similarly to a standard LFS cleaner, which has up to an approximate 34% decrease in performance [30]. The S4 cleaner is slightly more intrusive, degrading performance by approximately 50% in the worst case. The greater degradation is attributed mainly to the additional reads necessary when cleaning objects rather than segments. In addition, the S4 cleaner has not been tuned and does not include known techniques for reducing cleaner performance problems [21].

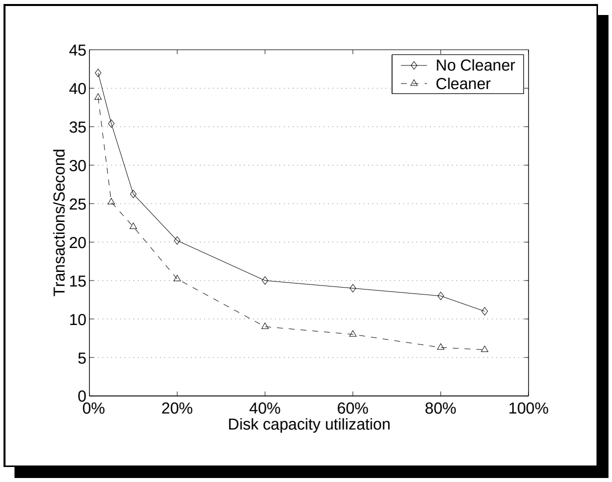

Figure 5: **Overhead of foreground cleaning in S4** – This figure shows the transaction performance of S4 running the PostMark benchmark with varying capacity utilizations. The solid line shows system performance on a system without cleaning. The dashed line shows system performance in the presence of continuous foreground cleaner activity.

5.1.4 Overhead of the S4 audit log

In addition to versioning, self-securing storage devices keep an audit log of all connections and commands sent to the drive. Recording this audit log of events has some cost. In the worst case, all data written to the disk belongs to the audit log. In this case, one disk write is expected approximately every 750 operations. In the best case, large writes, the audit log overhead is almost non-existent, since the writes of the audit log blocks are hidden in the segment writes of the requests. For the macro-benchmarks, the performance penalty ranged between 1% and 3%.

For a more focused view of this overhead, a set of micro-benchmarks were run with audit logging enabled and disabled. The micro-benchmarks proceed in 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.

Figure 6 shows the results. The create and delete phases exhibit a 2.8% and 2.9% decrease in performance, respectively, and the read phase exhibits a 7.2% decrease in performance. Read performance suffers a larger penalty because the audit log blocks become interwoven with the data blocks in the create phase. This reduces the number of files packed into each segment, which in turn increases the number of segment reads required.

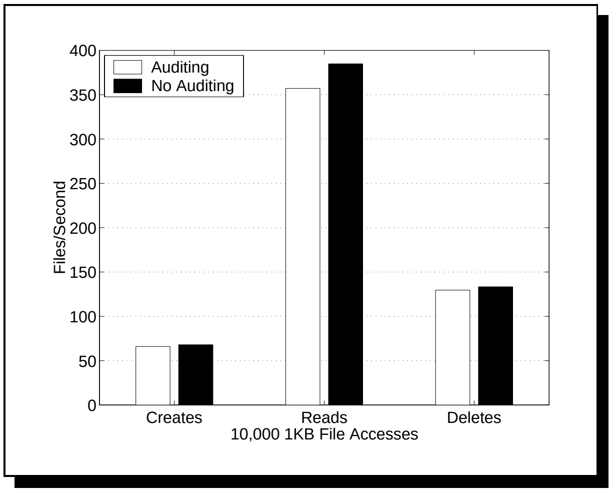

Figure 6: **Auditing Overhead in S4** – This figure shows the impact on small file performance caused by auditing incoming client requests.

5.1.5 Fundamental performance costs

There are three fundamental performance costs of self-securing storage: versioning, auditing, and garbage collection. Versioning can be achieved at virtually no cost by combining journal-based metadata with the LFS structure. Auditing creates a small performance penalty of 1% to 3%, according to application benchmarks. The final performance cost, garbage collection, is more difficult to quantify. The extra overhead of S4 cleaning in comparison to standard LFS cleaning comes mainly from the difference in utilized space due to the history pool.

The worst-case performance penalty for garbage collection in S4 can be estimated by comparing the cleaning overhead at two space utilizations: the space utilized by the active set of objects and the space utilized by the active set combined with the history pool. For example, assume that the active set utilizes 60% of the drive's space and the history pool another 20%. For PostMark, the cleaning overhead is the difference between cleaning performance and standard performance seen at a given space utilization in Figure 5. For 60% utilization, the cleaning overhead is 43%. For 80% utilization, it is 53%. Thus, in this example, the extra cleaning overhead caused by keeping the history pool is 10%.

There are several possibilities for reducing cleaner overhead for all space utilizations. With expected detection windows ranging into the hundreds of days, it is likely that the history pool can be extended until such a time that the drive becomes idle. During idle time, the cleaner can run with no observ-

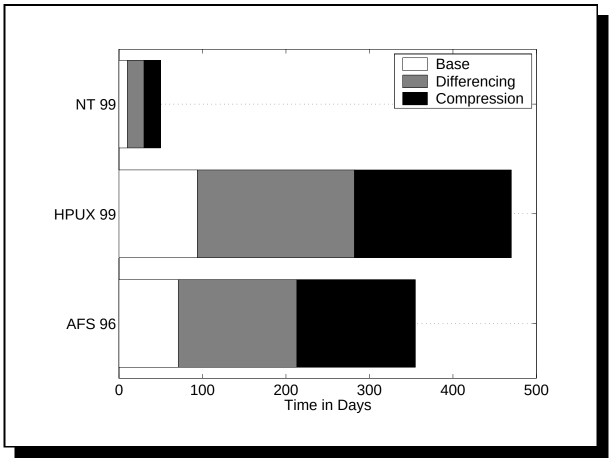

Figure 7: **Projected Detection Window** – The expected detection window that could be provided by utilizing 10GB of a modern disk drive. This conservative history pool would consume only 20% of a 50GB disk's total capacity. The baseline number represents the projected number of days worth of history information that can be maintained within this 10GB of space. The gray regions show the projected increase that cross-version differencing would provide. The black regions show the further increase expected from using compression in addition to differencing.

able overhead [2]. Also, recent research into technologies such as freeblock scheduling offer standard LFS cleaning at almost no cost [18]. This technique could be extended for cleaning in S4.

5.2 Capacity Requirements

To evaluate the size of the detection window that can be provided, three recent workload studies were examined. Figure 7 shows the results of approximations based on worst-case write behavior. Spasojevic and Satyanarayanan's AFS trace study [32] reports approximately 143MB per day of write traffic per file server. The AFS study was conducted using 70 servers (consisting of $32,000$ cells) distributed across the wide area, containing a total of 200GB of data. Based on this study, using just 20% of a modern 50GB disk would yield over 70 days of history data. Even if the writes consume 1GB per day per server, as was seen by Vogels' Windows NT file usage study [34], 10 days worth of history data can be provided. The NT study consisted of 45 machines split into personal, shared, and administrative domains running workloads of scientific processing, development, and other administrative tasks. Santry, et al. [29] report a write data rate of 110MB per day. In this case, over 90 days of data could be kept. Their environment consisted of a single file system holding 15GB of data that was being used by a dozen researchers for development.

Much work has been done in evaluating the efficiency of differencing and compression [3, 4, 5]. To briefly explore the potential benefits for S4, its code base was retrieved from the CVS repository at a single point each day for a week. After compiling the code, both differencing and differencing with compression were applied between each tree and its direct neighbor in time using Xdelta [19, 20]. After applying differencing, the space efficiency increased by 200%. Applying compression added an additional 200% for a total space efficiency of 500%. These results are in line with previous work. Applying these estimates to the above workloads indicates that a 10GB history pool can provide a detection window of between 50 and 470 days.

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 temporarily store exploit tools without fear that they will be recovered.

Versioning vs. snapshots: Self-securing storage can be implemented with frequent copy-on-write snapshots [11, 12, 17] instead of versioning, so long as snapshots are kept for the full detection window. Although the audit log can still provide a record of what blocks are changed, snapshots often will not allow administrators to recover short-lived files (e.g., exploit tools) or intermediate versions (e.g., system log file updates). Also, legitimate changes are only guaranteed to survive malicious activity if they survive to the next snapshot time. Of course, the potential scope of such problems can be reduced by shrinking the time between snapshots. The comprehensive versioning promoted in this paper represents the natural end-point of such shrinking—every modification creates a new snapshot.

Versioning file systems vs. self-securing storage: Versioning file systems excel at providing users with a safety net for recovery from accidents. They maintain old file versions long after they would be reclaimed by the S4 system, but they 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 long-term landmark versioning [28], recovery from users' accidents could be enhanced while also maintaining the benefits of intrusion survival.

Self-securing storage for databases: Most databases log all changes in order to protect internal consistency in the face of system crashes. Some institutions also retain these logs for long-term auditing purposes. All information needed to understand and recover from malicious behavior can be kept, in database-specific form, in these logs. Self-securing storage can increase the post-intrusion recoverability of database systems in two ways: (1) by preventing undetectable tampering with stored log records, and (2) by preventing undetectable changes to data that bypass the log. After an intrusion, self-securing storage allows a database system to verify its log's integrity and confirm that all changes are correctly reflected in the log—the database system can then safely use its log for subsequent recovery.

Client-side cache effects: In order to improve efficiency, most client systems use caches to minimize storage latencies. This is at odds with the desire to have storage devices 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 with capturing exploit tools, since they may never be written to the drive. Although client cache effects may obscure some of the activity in the client system, data that are stored on a self-securing storage device are still completely protected.

Object-based vs. block-based storage: Implementing a self-securing storage device with a block interface adds several difficulties. Since objects are designed to contain one data item (file or directory), enforcing access control at this level is much more manageable than attempting to 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.

Multi-device coordination: Multi-device coordination is necessary for operations such as striping data or implementing RAID across multiple self-securing disks or file servers. In addition to the coordination necessary to ensure that multiple copies of data are synchronized, recovery operations must also coordinate old versions. On the other hand, clusters of self-securing storage devices could maintain a single history pool and balance the load of versioning objects. Note that a self-securing storage device containing several disks (e.g., a self-securing disk array) does not have these issues. Additionally, it has the ability to keep old versions and current data on separate disks.

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., provide 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 detection window), and it uses Elephant's time-based access. The primary advantage of S4 over such systems is that it has been partitioned from client operating systems. While this creates another layer of abstraction, it adds to the survivability of the storage.

A self-securing disk drive would be another instance of many recent "smart disk" systems [1, 8, 15, 26, 35]. 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 intrusion recovery is to keep a periodic backup of files on trusted storage. Several file systems simplify this process by allowing a snapshot to be taken of a file system [11, 12, 17]. This snapshot can then be backed-up with standard file system tools. 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 long-existing critical data, the window of time in which data can be destroyed or tampered with is much larger than in S4—often 24 hours or more. Also, these systems are generally reliant upon a system administrator for operation, with a corresponding increase in cost and potential for human error. In addition, intrusion diagnosis is extremely difficult in such systems. Permanent file storage [25] provides an unlimited set of puncture-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 [27]. Many file systems use journaling to improve performance while maintaining disk consistency [6, 31, 33]. However, these systems delete the journal information once checkpoints ensure that the corresponding blocks are all on disk. S4's journal-based metadata persistently stores 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 existing 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.

Acknowledgments

We thank Brian Bershad, David Petrou, Garth Gibson, Andy Klosterman, Alistair Veitch, Jay Wylie, and the anonymous reviewers for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including CLARiiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. We also thank IBM Corporation for supporting our research efforts. This work is partially funded by the National Science Foundation via CMU's Data Storage Systems Center and by DARPA/ISO's Intrusion Tolerant Systems program (Air Force contract number F30602-99-2-0539-AFRL). Craig Soules is supported by a USENIX scholarship.

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] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Annual USENIX Technical Conference* (New Orleans), pages 277–288. Usenix Association, 16–20 January 1995.

[3] Randal C. Burns. *Differential compression: a generalized solution for binary files.* Masters thesis. University of California at Santa Cruz, December 1996.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. *ACM Symposium on Operating System Principles* (Austin, Texas, 8–11 November 1987). 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 Appliance, 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] 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.

[17] 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.

[18] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Deigo, CA, 23–25 October 2000). ACM, October 2000.

[19] Josh MacDonald. *File system support for delta compression*. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 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] *Object based storage devices: a command set proposal*. Technical report. October 1999. http://www.T10.org/.

[25] 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.

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

[27] 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.

[28] Douglas J. Santry, Michael J. Feeley, and Norman C. Hutchinson. Elephant: the file system that never forgets. *Hot Topics in Operating Systems* (Rio Rico, AZ, 29–30 March 1992). IEEE Computer Society, 1999.

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

[34] 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.

[35] 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.

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