Architecture and Interface of a Self-Securing Object Store

John D. Strunk
Electrical and Computer Engineering
Carnegie Mellon University
(jstrunk@andrew.cmu.edu)

May 8, 2000

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 this window, system administrators have valuable information for intrusion diagnosis and recovery. This thesis discusses the architecture behind the Self-Securing Storage Systems (S4) project. It also presents the external interface that the S4 drive uses for communication with client systems. The design and evaluation of the initial S4/NFS client is presented as an example client-side interface to a self-securing object store.

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 [18] or Adaptec's GoBack [4]) 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 the interface to our implementation, 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 30–100 GB drives for far longer than a week [18, 21]. Other challenges include achieving secure administrative control and dealing with denial-of-service attacks.

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 discusses the interface available to clients for data storage. Section 5 describes an NFS file system that has been built on S4. Section 6 contains a discussion of related work, and section 7 proposes future work for self-securing storage systems. 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 [2], 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 [13]. 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 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. 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 stored there. 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 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 cannot be imperceptibly altered. In addition, since the drive maintains these old versions, they can quickly be restored to their pre-intrusion state. 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 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.

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 access 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, 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. By making sure any given access is bound to a {client, user} pair, a self-securing storage device can assure the following:

- For an uncompromised client, accesses 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 [3, 5]. One solution would be the use of a network-level mechanism like IPSec [12], 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 object 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 files 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, recent analysis of workloads 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 obviously 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 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 remove 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 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 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 [18], 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.

4 S4 External Interface

S4 is an implementation of a network-attached self-securing storage device [6, 19, 20]. It exports an object interface to a set of clients via the network, and is designed to provide them with secure, versioned storage.

4.1 Provided Services

The basic S4 system was designed to provide a minimal but versatile set of services o that clients and servers could add upon the basic functionality and guarantees to create full-function storage systems without having to deal with overly complex or unnecessary functionality.

The basic S4 drive provides storage in the form of variable-sized objects. These objects exist in a flat namespace on the drive, and it is the responsibility of a higher level system to arrange these objects into a hierarchal structure if so desired. The drive also provides a method of "naming" objects for use as initial "mount points" in lieu of having special well-known objects. The S4 drive internally versions all data objects, their attributes, and ACLs. The list of named objects (a.k.a the partition table) is also versioned. All versioning is done transparently to client systems. In addition, the S4 drive enforces access control on the objects that it stores.

Any functionality beyond that mentioned above is the responsibility of higher-level storage management systems. For instance, S4 does not support concurrency control or locking of objects. Also, S4 does not attempt to interpret the contents of objects, so it has no method for maintaining file system integrity beyond per-object versioning and access auditing.

The S4 drive was designed with efficiency as a top priority. One of the project goals was to provide this high level of data protection for little or no performance cost. Recent performance analysis of S4 (with the S4/NFS client) shows that the it has comparable performance characteristics to BSD's NFS server [20].

4.2 Access control

Access control verification is an integral part of the S4 system because it enables the access of intruders and nosy users to be limited to the minimum possible set of objects, but most importantly, it allows proper logging and auditing of accesses so that the system administrator has the maximum possible amount of information when trying to diagnose and recover from an intrusion. The security for the S4 system has two parts. The first part is concerned with the integrity and privacy of requests to the storage system, and the other is the access control mechanisms that are applied to all requests.

The authenticity, integrity, and privacy are implemented as two distinct parts. The first is a network-level mechanism such as IPSec that can be implemented in hardware thus causing minimal performance impact. This portion of the security should provide all three of these guarantees for client to drive communication. The other piece to authenticating accesses is to verify a token that is provided with each request. This token serves as proof that the userid associated with the request is the proper user's identity. This is designed to prevent a compromised client from having access to all information stored on the S4 drive. At a minimum, it limits the access of a client to the access of the recent users of that client.

Once requests are authenticated, the drive checks the access permission of the {client, user} pair by examining the ACL that is associated with the object named in the request.

Table 4.2 shows a list of the ACL fields that are supported by the S4 drive. Based on these ACL fields, users are assigned permissions both for individual objects and for the drive as a whole. For recovery from intrusions, system administrators would need to authenticate as a user that has both `Administrator` and `Recovery` permission for the drive. This will permit them to recover all necessary objects from the history pool, as well as read and write the current object versions. Another aspect of the ACL flags worth noting is that by clearing the `Recoverable` flag, normal users can make object versions unrecoverable. Any users with the drive `Recovery` ability can still recover these versions, however.

ACL flag	Description
— disk ACLs —	
Administrator (A)	Permits the user to add and delete users on the device; Also allows the user full access to the current objects on the drive.
Create (C)	Permits the user to create new objects on the device
Partition (P)	Permits the user to create/delete name to object associations
Recover (V)	Permits the user to recover all versions of all objects in the history pool
— object ACLs —	
Administrator (A)	Permits the user to modify the ACL of this object
Read (R)	Permits the user to read the data stored in this object
Write (W)	Permits the user to write or truncate the data stored in this object
Append (P)	Permits the user to append data to the end of the object; This is redundant if the user also has Write access
Delete (D)	Permits the user to delete this object
Recoverable (V)	Permits the user to recover *this version* of the object from the history pool
Lookup (L)	Permits the user to retrieve the attributes and ACL for the object

Table 1: **ACL fields supported by the S4 drive**: The ACL fields are associated with S4 `userIDs` on either a per drive or per object basis. The security subsystem uses these ACLs to make access control decisions.

4.3 RPC interface

Since the S4 prototype implementation is a network-attached object store, clients must access the drive via an RPC interface. Tables 4.3 & 4.3 list each type of RPC operation that is supported by the drive.

This interface was designed to provide a complete, yet simple interface to the S4 drive. Although not shown in the table, each command also includes cryptographic authentication information so that the drive front end may verify the access permission of the client and user making the request.

4.4 History interface

Tables 4.3 & 4.3 indicate which RPC commands may be used for accessing objects in the history pool in addition to accessing the current version. No commands that modify objects are supported for this "in-time" access of objects in the history pool since the it is considered to be a read-only, however all commands that retrieve information accept a `time` field to indicate which version of the object to read. The drive will automatically clean old objects from the history pool once they

RPC Operation	In Time?	Description
`read(objID, offset, length)`	Y	Reads `length` bytes from an object starting at `offset`
`write(objID, data, offset, length)`	N	Writes `length` bytes starting at `offset`
`append(objID, data, length)`	N	Appends `length` bytes to the end of the object
`Truncate(objID, length)`	N	Truncates the size on an object to `length`
`create()`	N	Creates a new object and returns its `objID`
`delete(objID)`	N	Deletes the object from the current set of objects
`getattr(objID)`	Y	Retrieves the S4 specific and the opaque attributes associated with an object
`setattr(objID, attrs)`	N	Writes new opaque attributes for the specified object
`getaclbyindex(objID, index)`	Y	Retrieves an ACL entry based on an index into the list of entries
`getaclbyuser(objID, userID)`	Y	Retrieves an ACL entry for a given user
`setacl(objID, acl)`	N	Sets an ACL entry for an object

Table 2: **Object commands**: RPC commands for manipulating objects. Commands which support "in time" access can accept an additional time parameter to access old object versions from the history pool

are outside of the detection window.

The main method for recovering from accidents and intrusions is to *copy-forward* the desired version from the history pool, writing it as the new version of the object. The initial design of S4 included a copy-forward command, but it was later determined to be unnecessary since a client with proper access may `read` an old version from the history pool and `write` it as the current version of an object. Thus this two step process is the main method used to recover old versions of objects.

4.5 Optimizations

During initial testing, the necessity for certain optimizations became apparent. The client implementation was generating a very large number of attribute retrieval and modification requests.

RPC Operation	In Time?	Description
— Administrative operations —		
`setwindow(time)`	N	Sets the guaranteed size of the recovery window
— General operations —		
`sync()`	N	Flushes cached writes to disk; This RPC does not return until the synchronization is complete
— Partition operations —		
`plist()`	Y	Retrieves the list of named objects
`pmount(name)`	Y	Returns the `objID` associated with the given name
`pcreate(objID, name)`	N	Associates `name` with the specified object
`pdelete(name)`	N	Remove a `name` to `objID` association

Table 3: **Miscellaneous commands**: RPC commands for administration and manipulating object names

These requests were due to updating file system timestamps. The optimization to minimize this traffic allows combining of RPCs so that each data modification RPC (`create`, `read`, `write`, `append`, and `truncate`) can also perform a `getattr` and `setattr` on the object being modified. The data operation is performed first, followed by the `setattr`, and finally the `getattr`. This allows the new file system attributes to be stored upon successful completion of the data operation, and the resulting new attributes are returned to the client to keep its cache fresh.

A second optimization was made to allow retrieval and modification of up to eight ACL entries at a time on an object. This also greatly decreased the number of RPCs that were necessary by allowing the client to make batch modifications to the ACL entries for an object. Both of these optimizations are handled within the front end module of the drive by splitting the incoming RPC operation into multiple object system operations.

5 S4 NFS Client

The first client file system developed for use with the S4 system was an S4/NFS client. This client is a user-level process that translates NFS requests made by the local machine's kernel into S4 RPC calls and relays them to the drive. The S4 client implements the NFS version 2 specification, and appears to the local machine as an NFS server. The local machine mounts this NFS "server" as it would any other NFS server, except that the network communication to the S4 client is over the loopback interface to ensure secure communication and not expose the S4 client to external network traffic.

5.1 Object overview

Once the S4 client receives an NFS request, it translates the file/directory operation into RPC operations on S4 objects. When implementing the S4/NFS file system, there are two types of objects that the client uses—directory objects and file objects. It is important to note that while the client makes a distinction between these two object types, the drive does not. These two types of objects plus the S4 attributes and the opaque attributes are sufficient to implement the S4/NFS file system. The NFS protocol identifies files and directories using a filehandle. S4 `objectID`s can be directly hashed into NFS filehandles and vice versa.

5.2 NFS Attributes

The NFS file system maintains a set of attributes for each file and directory. This attribute structure contains information such as the file type, owner, group, and UNIX access permissions. This structure is stored directly in the opaque space provided by the S4 system, with the exception of two fields. The `size` field of the NFS attribute structure is generated directly from the S4 `size` attribute, which is an accurate measure of the size of the file or directory being stored. The `fileid` field, a unique identifier for each file/directory on an NFS server, is generated from the lower 32 bits of the object's ID. Generating the `fileid` field when the attributes are read instead of storing it in the opaque space was chosen as an optimization that allowed the `setattr` and `create` S4 RPC calls to be combined.

Since nearly all operations access this attribute structure, the client maintains an attribute cache that can be indexed by its S4 `objectID`. This cache is checked for a valid entry when the low-level S4 client `getattr` is called. If the attribute structure is found in the cache, it is returned instead of generating an RPC. This cache uses a FIFO replacement policy once the cache fills, and also frees attributes from the cache when they reach a certain age. The current client cache configuration will maintain the attributes for 2000 objects and invalidate them when they have aged for 60 seconds. The cache is kept fresh by utilizing the previously discussed RPC combining to issue a `getattr` request on each data RPC. This ensures that the attribute cache is always fresh for frequently accessed objects. With the cache as described, miss rates are below 1% for most benchmarks. For instance when building the S4 source tree, the miss rate for the attribute cache is .75%.

5.3 Directory structure

NFS directories are packed into objects as a set of records that associate file names with filehandles. Initially, the records were fixed size entries that held the ASCII name of each entry in the directory along with the NFS filehandle of the associated entity to which it referred. This created a problem for directory operations that needed to remove entries. In order to remove an entry, the client had to find the offset into the object containing the record to remove, copy the last record in the object to this position (overwriting the entry to remove), then truncate the directory object by the size of one record. This required multiple RPCs to implement, and posed a very realistic possibility of directory corruption if multiple clients attempt to modify the directory simultaneously.

The structure of the directories was then changed to use a log-structured approach and variable size records. The log structure reduces all normal directory operations to a single append operation, and by using variable sized records, directory objects can be over 8x smaller. The space savings arises because, when using fixed size records, a record must be able to support the maximum length file name (255 bytes) plus the space for an NFS filehandle (32 bytes). By making the records variable size and storing only as many bytes for the name as are needed (plus one byte for a terminator), the size of the average record could fall to only 43 bytes (assuming an average of 10 characters in the filename) from 287, which is a saving of 6x.

The log structure permits changes to the directory by appending a record onto the end of the object. The last record that contains a given filename takes precedence, and the NFS filehandle consisting of all zeros indicates a deleted name. Based on this, to set a link, an append of the new record is sufficient to handle this, and removal of a link is an addition of a record consisting of the filename associated with an all zero filehandle. This also permits operations such as rename within the same directory to be implemented as a single operation as well (by appending two records instead of the usual one).

Unfortunately, this log structure causes the size of a directory to grow with each operation, so it is necessary to clean a directory periodically. When clients read the directory, they keep a count of the number of records encountered in the object, and the number of actual links in the directory. When these numbers pass a certain efficiency threshold (e.g. 10x as many records as entries, or 100 more records than entries), the client attempts to clean the directory. Since the client has just finished reading the directory, it has a full list of the directory's contents, so it truncates the directory to a length of zero (removing all previous records), then writes out one

NFS Operation	S4 RPC calls		Description
	minimum	average	
`getattr`	0	.0071	Retrieves the attribute structure of a file or directory
`setattr`	2.5	3.0236	Sets the attribute structure of a file or directory
`lookup`	0	.0043	Returns the NFS filehandle for a given name in a directory
`readlink`	1	1.0000	Reads the pathname stored in a symbolic link
`read`	1	1.0018	Reads the contents of a file
`write`	2	2.0000	Writes data into a file
`create`	4	6.0034	Creates a new file in a directory
`remove`	3	3.0012	Removes a file from a directory
`rename`	4–5	6.0000	Renames a file
`link`	3	3.0000	Makes a hard link to a file
`symlink`	5	7.0000	Creates a symbolic link to a file or directory
`mkdir`	5	12.0000	Creates a new directory
`rmdir`	3	9.0000	Removes an empty directory
`readdir`	1	2.6678	Returns the contents of a directory

Table 4: **NFS RPC operations**: This table lists the major NFS version 2 procedures and the number of S4 RPC calls needed to implement the operation with the S4/NFS client. The minimum number of operations is based on performing the optimal set of operations and having both client caches maintain a 100% hit rate. The column labeled "average" shows the average number of RPC operations actually performed for each NFS operation during a CVS checkout and build of the S4 source tree. Some operations differ significantly from the minimum because of inefficiencies in the client implementation.

record for each name in its list. This cleaning operation requires only two RPC calls and need not be performed very often. Only clients with permission to write to a directory object are able to clean it. The switch to a log-structured directory decreased the likelihood of directory corruption due to multiple simultaneous updates since these updates are now `append` operations instead of `write` and `truncate`. The cleaning, however is still susceptible to this.

The client maintains a directory cache (currently 200 directories for a maximum of 60 seconds) to speed NFS `lookup` and `readdir` operations as well as lower level directory manipulation operations. This cache is also very effective, having a miss rate of .45% while building the S4 source tree.

5.4 S4/NFS efficiency

In order to implement the NFS file system on top of S4, some NFS operations require multiple object manipulations. Table 5.4 shows a list of the NFS RPC operations, a description of the operation, the minimum number of RPCs that can be used to implement that operation (with the construction of objects as described), and the average number of RPCs performed during a CVS checkout and build of the S4 source tree. The client has a number of inefficiencies that are introduced due to its construction. These prevent the actual number of RPCs for the client from approaching the lower bound.

At the end of each NFS operation that modifies state on the drive, a `sync` RPC is required in order to support NFS v2 semantics of all data being in non-volatile storage when the call returns.

5.5 S4/NFS shortcomings

Since the S4 drive only has knowledge of objects, and not the relationships between objects, problems arise in implementing a file system on top of the objects. Users that have permission to create objects (files) on the drive have the potential to create "detached" objects. These are objects that are valid in the sense of S4, but are not linked into the file system directory structure. These objects would be inaccessible to users who could only navigate the file system's directory structure, but could provide covert storage for a user/client that is willing to directly manipulate objects. These detached objects can also be created accidently by client crashes during an NFS create operation. There is no way for the S4 system to force all objects to be a part of the directory structure because the drive does not understand that structure, and there is no middle-man to supervise clients' access.

In addition to the detached object problem, there are several other problems that arise from the use of ACLs for determining access permissions on each object individually. In standard UNIX file systems, users may rename, link, or unlink a file in any directory for which they have write permission, and they need not have any permission to access the file. Normal file systems have complete access to the storage device, and act as trusted entities to update metadata such as the file's link count on behalf of the user. In S4, a file's metadata is stored in the opaque attribute space of the file's object, and is subject to the restrictions of the ACL for that file. When a link is made to a file, the link count must be incremented, but in S4, this may not be possible if the user performing the link only has permission to write to the directory, but not to modify the file's attributes. In addition, file systems delete files when there are no more names that reference that data. In S4, if the user who removes the final name for a file does not have delete permission for the file's object, it will become a detached object on the drive.

5.6 File system cleaner

Conventional systems have a file server between clients and drives to enforce file system integrity and perform privileged operations on behalf of users. This prevents the problems discussed above. NASD, like S4, exposes an object interface to clients, but for the NASD NFS client, only `getattr`, `read`, and `write` operations are sent directly to the disks. All other NFS operations were sent to the NASD file manager for processing on behalf of the client. This ensures that as long as the file manager was intact, the file system was maintained in a consistent state.

S4 removes the file manager since it is a potential source of intrusion into the system, but due to the lack of an overseeing entity, the file system may become inconsistent. To combat this, it is possible to have a background cleaner that repairs objects at the file system level. This entity would be able to run during non-peak times and clean up deleted objects and link counts that clients were unable to maintain themselves. Also, since this cleaner is a separate entity, and not performing actions on behalf of users, it would be easy to audit its access and undo any actions that the cleaner would make if it were to be compromised.

6 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 [15, 7]. Santry, et. al, provides a good discussion of techniques for traversing versions and deciding what to retain [18]. 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 such systems as these is that it has been partitioned from the operating system. While this creates another layer of abstraction, it adds to the survivability of the storage.

S4's device-embedded storage management is another instance of many recent "smart disk" systems [1, 3, 11, 17, 22]. All of these exploit the increasing computational 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 attempting to create a new standard for object-based storage devices. The S4 interface is similar to this proposal.

The standard method of intrusion 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 [8, 9, 14]. This snapshot can then be accessed through the standard file system tools for file retrieval. Spiralog [10] 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 tampered with or destroyed 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 [16] 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.

7 Future Work

With an initial version of the S4 system in place, several major areas of interest still remain. The initial implementation of S4 is lacking several pieces of the overall architecture. This section describes several of the remaining open issues for the S4 implementation, and future directions of research on the project.

7.1 Security

The security subsystems of the S4 implementation have not yet been built, though the infrastructure to support this is in place. Access control lists are maintained for each object on the drive, and the S4/NFS client adjusts these to best fit the UNIX mode bits for the "owner" and "other" entities. The drive security module that will verify permission according to the ACL flags needs to be implemented. In addition, the client needs to be modified to provide tokens according to the user accessing the drive, and facilities need to be implemented for key management within the drive.

Logging of accesses also needs to be added to the drive. It is a critical part of intrusion analysis and recovery to have a log of all accesses that can be analyzed based on both users and client machines. When implementing logging and access control, it may prove to be a challenging task to record this information while not noticeably impacting system performance. The ACL lookup on the drive has been optimized for efficiency, but the true impact on the system will only be known when it is actually running.

Another interesting area of further work is in preventing the history pool from overflowing. Since disk space is finite, the drive needs to monitor and take steps to avoid being faced with exhaustion of free space. Since ceasing to version objects or prematurely reclaiming objects from the history pool presents a security hole, further work is needed on ways to prevent the space exhaustion as opposed to recovering once it occurs.

7.2 Recovery and analysis tools

Providing tools for recovering and analyzing old object versions is a daunting task. The amount of information that will be available could quite easily be overwhelming to a system administrator. Simple recovery tools, such as those used for recovery from accidents, will be straight forward since the user is expected to already know the exact data item he wishes to retrieve.

Implementation of utilities to recover from intrusions will be much more difficult. They must correlate the historical objects with the access logs and present this in a form that the system administrator can use. The complete picture may be obscured by client-side caches (both read and write), and it will be extremely difficult to track the propagation of tainted data through the system.

7.3 Other file systems

The S4 system will also benefit from having more types of client file systems implemented over top of the object store. Every effort has been made to ensure that the interface is general and versatile enough to implement a very wide range of file systems, but this warrants further investigation. The NFS client implements a file system with access control based on the familiar UNIX permissions. Implementation of other clients such as an AFS client (which uses ACLs) or a client for a database system can lead to valuable insight about the long term feasibility of the object-based interface to self-securing storage devices.

7.4 Transactions

Introducing the notion of transactions or conditional writes may provide a way for clients to avoid accidental corruption of objects when multiple clients access a single object. Currently, any locking or transaction mechanism must be implemented at a higher level, and this can lead to performance degradation or security holes.

It should be possible to support some limited form of transactions within the drive that utilizes the history pool as a way of aborting transactions that should not be committed. It would also be straight forward to implement a conditional write primitive that could condition a write on the current version's modification time.

7.5 Multi-device coordination

Groups of S4 devices could be assembled to present one, large, object storage image. Regardless of the distribution of data across these devices, correlation of changes to object data would be more difficult than in a conventional system. The recovery tools would have to assemble and correlate the information from all the devices to recover objects from the history pool. This would be particularly difficult for data that is striped across multiple devices. Pools of devices may be able to benefit from load balancing or sharing the history pool across the cluster.

8 Conclusion

The self-securing storage paradigm promises to raise the level of security available to users by making the storage device responsible for maintaining the integrity of data. The S4 implementation has shown that this level of intrusion tolerance (into clients and servers) and convenience can be provided at very little performance cost.

The S4 interface provides a complete set of operations necessary to implement a file system on top of the object storage provided by the drive. The S4/NFS client implementation has shown that even though multiple operations are required to implement high-level file system operations, it can be handled in an efficient manner.

The S4 project still has a few pieces which need to be incorporated, and there are many interesting research questions that have yet to be explored.

9 Acknowledgements

Thanks to the other members of the S4 team (Greg Ganger, Garth Goodson, Mike Scheinholtz, and Craig Soules) who have helped immensely and dedicated long hours to make this project a reality. Thanks also to my advisor, Greg Ganger, who had the utmost patience while I decided what area of computer systems interests me. Also, a special thanks to my wife, Corley, for her support and understanding.

Thanks to 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. I also thank IBM Corporation and DARPA's Information Technology Office for supporting my research efforts.

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

[3] 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.

[4] GoBack. http://www.goback.com/.

[5] 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.

[6] Garth R. Goodson. *I/O infrastructure support for network-attached storage devices*. Master's thesis. Electrical and Computer Engineering Department, Carnegie Mellon University, May 2000.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] Michael Scheinholtz. *An efficient versioning file system for self-securing storage*. Master's thesis. Electrical and Computer Engineering Department, Carnegie Mellon University, May 2000.

[20] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. *Design and implementation of a self-securing storage device*. TR CMU-CS-00-129. May 2000.

[21] 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.

[22] 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.