Survivable Storage Systems

Gregory R. Ganger, Pradeep K. Khosla, Mehmet Bakkaloglu, Michael W. Bigrigg, Garth R. Goodson, Semih Oguz, Vijay Pandurangan, Craig A. N. Soules, John D. Strunk, Jay J. Wylie
Carnegie Mellon University

Abstract

Survivable storage systems must maintain data and access to it in the face of malicious and accidental problems with storage servers, interconnection networks, client systems, and user accounts. These four component types can be grouped into two classes: server-side problems and client-side problems. The PASIS architecture addresses server-side problems, including the connections to those servers, by encoding data with threshold schemes and distributing trust amongst sets of storage servers. Self-securing storage addresses client and user account problems by transparently auditing accesses and versioning data within each storage server. Thus, PASIS clients use threshold schemes to protect themselves from compromised servers, and self-securing servers use full access auditing to protect their data from compromised clients. Together, these techniques can provide truly survivable storage systems.

1. Introduction

As society increasingly relies on digitally stored and accessed information, supporting the availability, persistence, integrity, and confidentiality of this information becomes more and more crucial. We need storage systems to which users can entrust critical information, ensuring that it persists, is continuously accessible, cannot be destroyed, and is kept confidential. Further, with the continuing shift towards pervasive computing and less-expert users/administrators, information storage infrastructures must be more self-sufficient. A *survivable storage system* provides these guarantees over time and despite failures and malicious compromises of storage nodes, client systems, and user accounts. Current storage system architectures fall far short of this ideal. We are exploring the potential and trade-offs of two complementary approaches to creating storage systems that survive problems with their components:

Surviving storage node problems: a storage system can survive failures and compromises of storage nodes by entrusting data to sets of nodes via well-chosen encoding and replication schemes. Many such schemes have been proposed and employed over the years, but little understanding exists of the large trade-off space that they comprise. The selection and parameterization of the data distribution scheme has a profound impact on the storage system's availability, security, and performance. Our focus is on developing solid understanding of these trade-offs and on developing automated approaches to configuring and reconfiguring these systems. We investigate the tradeoffs inherent to survivable storage in the context of the PASIS system. PASIS is a storage system that encodes information via threshold schemes so as to distribute trust amongst storage nodes in the system.

Surviving malicious user activities: until their presence is detected, there is no way to differentiate successful intruders from legitimate users. Therefore, any approach to surviving malicious user activities must consider all requests to be suspect. To address this need, we propose *self-securing storage* wherein storage nodes internally version all data and audit all requests for a guaranteed amount of time, such as a week or a month. This device-maintained history information prevents intruders from destroying or undetectably tampering with stored information. It also provides information for diagnosis and recovery from intrusions. We explore the feasibility and design space of self-securing storage with a prototype self-securing NFS server.

This paper describes our progress on these two halves of building survivable storage systems. Section 2 describes PASIS and how it uses threshold schemes on client systems to survive storage server and network problems. Section 3 describes self-securing storage and how it uses server-side resources to protect stored data from compromised client systems.

2. PASIS

Survivable systems operate from the fundamental design thesis that no individual service, node, or user can

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

be fully trusted: having some compromised entities must be viewed as the common case rather than the exception. Survivable storage systems, then, must replicate and distribute data across many nodes, entrusting its persistence to sets of nodes rather than to individual nodes. Individual storage nodes must not even be able to expose information to anyone; otherwise, compromising a single storage node would let an attacker bypass access-control policies.

To achieve survivability, storage systems must be decentralized and must spread information among independent storage nodes. Prior work in cluster storage systems (e.g., Berkeley's xFS [2], CMU's NASD [11], Compaq's Petal [17], and MIT's BFS [7]) provides much insight into how to efficiently decentralize storage services while providing a single, unified view to applications. A key open issue is how information should be spread across nodes.

Availability and confidentiality of information are primary goals of many storage systems. Most systems enhance availability by providing full replication, but a few systems employ erasure-resilient correction codes, which use less space. Client-side encryption can protect information confidentiality even when storage nodes are compromised. Threshold schemes—also known as secret-sharing or information dispersal algorithms—offer an alternative that provides both information confidentiality and availability in a single, flexible mechanism. These schemes encode, replicate, and divide information into multiple pieces, or shares, that can be stored at different storage nodes. The system can only reconstruct the original information when enough shares are available.

This section presents an overview of general threshold schemes and how they can be used to build a survivable storage system. The PASIS architecture, its characteristics, and the trade-offs it makes available are also explained. More details of the PASIS system can be found in [30].

2.1. General threshold schemes

PASIS uses *general threshold schemes* to encode data before it is stored. Specifically, a *p-m-n* general threshold scheme breaks data into *n* shares such that any *m* of the shares can reconstruct the original data and fewer than *p* shares reveal absolutely no information about the original data. Although encryption makes it difficult to ascertain the original data, it does not change the value of *p* since information is still available for theft (in an information-theoretic sense). Different parameter selections of *p-m-n* expose a large space of encoding mechanisms for storage.

An example of a specific threshold scheme is N-way replication. It is a 1-1-N threshold scheme. That is, each replica reveals information about the encoded data (p=1). A single replica is required to reconstruct the original data (m=1), and there are N replicas to select from when attempting to reconstruct the original data (n=N). Table 1 lists specific examples of general threshold schemes.

Table 1. Specific threshold schemes

Parameters	Description
1-1-n	Replication
1-n-n	Decimation (Striping)
n-n-n	Splitting (XORing)
1-m-n	Information Dispersal
m-m-n	Secret Sharing
p-m-n	Ramp Scheme

Threshold schemes can be implemented in different manners. Blakley's secret sharing scheme works in an *m*-dimensional space [3]. Secrets (data) are points in the space, and shares are multidimensional planes. Fewer than *m* shares represent a multidimensional plane that contains the secret. However, since all points in the field being considered are part of the plane, no information is revealed. With *m* shares, a single point of intersection—the secret—is determined. Figure 1 illustrates Blakley's secret sharing scheme. Shamir's secret-sharing scheme, developed at the same time as Blakley's, is based on interpolating the coefficients of a polynomial by evaluating the polynomial at certain points [25].

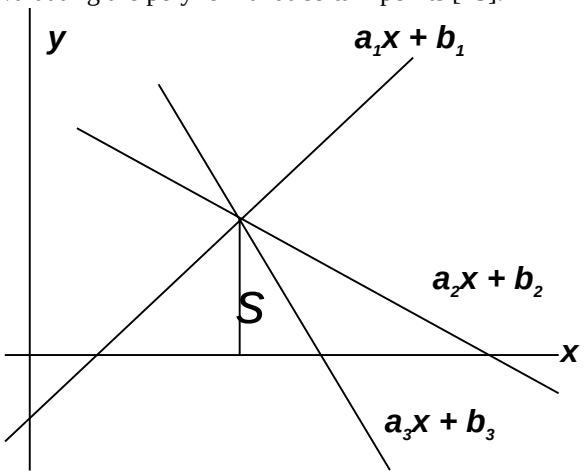

Figure 1. Blakley's secret sharing scheme with *m=2, n=3,* and original data *S*.

All implementations of secret sharing use random numbers to provide the guarantee that collecting fewer than *p* shares reveals no information about the original data. Rabin's information dispersal algorithm, a *1-m-n* threshold scheme, does not use random numbers [21]. Indeed, the parameter *p* indicates the number of random numbers required per encoding (i.e. *p*-1 random numbers

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

are required to encode a secret for any threshold scheme). Ramp schemes implement the full range of general *p-m-n* threshold schemes [8]. Ramp schemes operate like secret-sharing schemes up to *p* shares and like information dispersal schemes with *p* or more shares.

Threshold schemes can be used instead of cryptographic techniques to guarantee the confidentiality of information, and the two techniques can also be combined. For example, *short secret sharing* encrypts the original information with a random key, stores the encryption key using secret sharing, and stores the encrypted information using information dispersal [16]. Short secret sharing offers a different set of trade-offs between confidentiality and storage requirements than general threshold schemes. The confidentiality of the data stored by short secret sharing hinges on the difficulty of analyzing the information gained by collecting shares, because the information gained pertains to the encrypted data.

An extension to threshold schemes is cheater detection. In a threshold scheme that provides cheater detection, shares are constructed in such a fashion that a client reconstructing the original information object can tell, with high probability, whether any shares have been modified [28]. This technique allows strong information-integrity guarantees. Cheater detection can also be implemented using cryptographic techniques, such as adding digests to data before encoding or to the shares themselves after the data has been encoded.

2.2. PASIS architecture

The PASIS architecture, shown in Figure 2, combines decentralized storage systems, data redundancy and encoding, and dynamic self-maintenance to achieve survivable information storage. A PASIS system uses threshold schemes to spread information across a decentralized collection of storage nodes. Client-side agents communicate with the collection of storage nodes to read and write information, hiding decentralization from the client system.

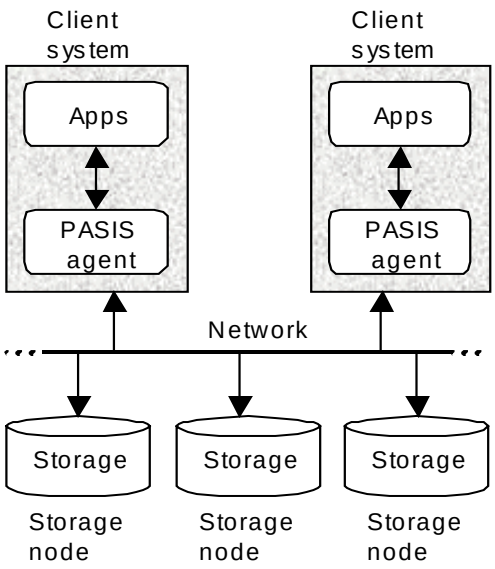

Figure 2. PASIS architecture

Several current and prior efforts employ the PASIS architecture to achieve survivability. For example, the Intermemory Project [12] and the Eternity Service [1] proposal use decentralization and threshold schemes for long-term availability and integrity of archived write-once digital information. The e-Vault [14] and Delta-4 [10] projects do the same for online read-and-write information storage. Delta-4 additionally addresses information confidentiality and other system services, such as authentication. All of these projects have advanced the understanding of these technologies and their roles in survivable storage systems, but such systems have not yet achieved widespread use.

2.3. PASIS system components and operation

A PASIS system includes clients and servers. The servers, or storage nodes, provide persistent storage of shares; the clients provide all other aspects of PASIS functionality. Specifically, PASIS client agents communicate with collections of PASIS servers to collect necessary shares and combine them using threshold schemes. This approach helps the system scale and simplifies its decentralized trust model. In fact, some system configurations can employ PASIS servers that are ignorant of decentralization and threshold schemes; for example, in our simplest demonstration, the PASIS client uses shared folders on Microsoft's Network Neighborhood as storage nodes. More advanced storage servers are clearly possible. Figure 3 presents the design of the PASIS client agent.

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

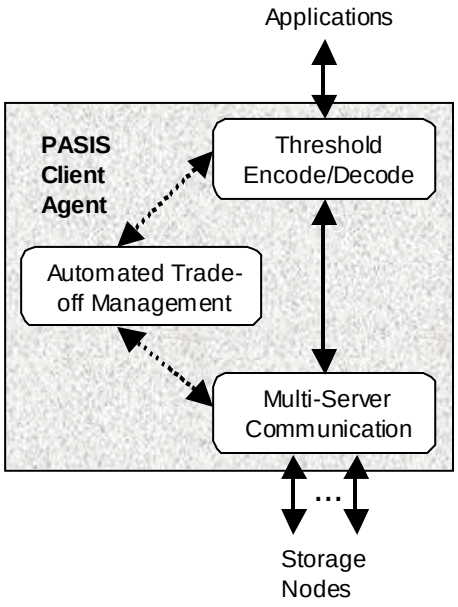

Figure 3. PASIS client agent

As with any distributed storage system, PASIS requires a mechanism that translates object names—for example, file names—to storage locations. A directory service maps the names of information objects stored in a PASIS system to the names of the shares that comprise the information object. A share's name has two parts: the name of the storage node on which the share is located and the local name of the share on that storage node. A PASIS file system can embed the information needed for this translation in directory entries. For example, our PASIS implementation of NFS (Network File System) functions in this way.

A *p-m-n* threshold scheme breaks information into *n* shares so that any *m* of the shares can reconstruct the information and fewer than *p* shares reveal no information. To service a read request, the PASIS client:

- Looks up, in the directory service, the names of the *n* shares that comprise the object.
- Sends read requests to at least *m* of the *n* storage nodes.
- Collects the responses. If it receives fewer than *m* responses, the client retries the failed requests. Alternatively, the client can send read requests to other previously unselected storage nodes. This step continues until the client has collected *m* distinct shares.
- Performs the appropriate decode computations on the received shares to reconstruct the original information.

Performing a write in PASIS is slightly different than performing a read. The write process does not complete until at least *n*–*m*+1 (or *m*, whichever is greater) storage nodes have stored their shares. That is, a write must ensure that fewer than *m* shares have not been overwritten to preclude reading *m* shares that were not updated. To maintain full availability, all *n* shares must be updated.

In addition, consider two clients, A and B, writing to the same object, D, concurrently. Because PASIS storage nodes are independent components of a distributed system, no assumptions should be made about the order in which the storage nodes see the writes. Thus, some storage nodes could have shares for D_A, while others could have shares for D_B. A similar problem arises if a client accidentally or maliciously updates only a subset of the shares.

One approach to handling concurrency problems assumes that a higher-level system addresses them. There are domains in which this approach is feasible. For example, information belonging to a single user and distributed applications that manage their own concurrency can use a storage system that does not guarantee correctness of concurrent writes.

For general-purpose use, a PASIS system must provide a mechanism that guarantees atomicity of operations. One such mechanism, atomic group multicast, guarantees that all correct group members process messages from other correct members in the same order. Atomic group multicast technology has been developed for several secure distributed systems, such as Rampart [22] and BFS [7]. Unfortunately, atomic group multicast can be expensive in large and faulty environments because the group members often must exchange many rounds of messages to reach agreement.

Because these technologies are believed to enhance storage system survivability, we have focused our efforts on designing survivable systems with performance and manageability on a par with simpler, conventional approaches. Only when such systems exist will survivable information systems be widely adopted.

2.4. PASIS architecture characteristics

The PASIS architecture can provide better confidentiality, availability, durability, and integrity of information than conventional replication—but at a cost in performance.

To compare the PASIS architecture to a conventional information storage system, consider a PASIS installation with 15 storage nodes that uses a 3-3-6 threshold scheme and uniformly distributes shares among storage nodes. The conventional installation, on the other hand, organizes 15 servers into five server groups, each storing 20 percent of the information on a primary server and two backup replica servers. A summary of the comparison

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

between a PASIS system and a conventional primary-backup system is given in Table 2.

Table 2. Summary of security comparison

Characteristic	PASIS	Primary-Backup
Confidentiality		
Percentage of information revealed if one storage node is compromised	0	20 percent
Percentage of information revealed if three storage nodes are compromised	4.4 percent	Up to 60 percent
Availability		
Probability that the system cannot serve a read request if each node fails with 0.001 probability	1.5×10^{-11}	10^{-9}
Durability		
Number of nodes that must be destroyed to erase a piece of information	4	3 (a server group)
Percentage of information erased when the above occurs	1.1 percent	20 percent
Integrity		
Nodes that must be compromised to falsely serve a read request	3 (*m* nodes)	1 (primary server)
Nodes that must be compromised to modify stored information without authorization	4 (greater[*n*-*m*+1,*m*])	1 (primary server)
Required Storage Blowup		
Secret sharing	6x (*n*x)	3x (*n*x)
Information Dispersal	2x (*n*x/*m*)	3x (*n*x)
Latency		
Reading small objects	Significantly higher latency for PASIS	
Reading large objects	Similar latencies	

Confidentiality determines a system's ability to ensure that only authorized clients can access stored information. To breach the conventional system's confidentiality, an intruder only needs to compromise a single storage node that has a replica of the desired object. In a PASIS system, an intruder must compromise several storage nodes to breach confidentiality.

Availability describes a system's ability to serve a specific request. For comparison purposes, assume that each storage node fails independently with a probability of 0.001. With this assumption, the PASIS system has two orders of magnitude higher availability than the conventional system.

Durability is a system's ability to recover information when storage nodes are destroyed. In the conventional system, an intruder must destroy three storage nodes (an entire server group) to maliciously erase information. In the PASIS system, an intruder must destroy four storage nodes to erase information.

Integrity is a system's ability to ensure that it correctly serves requests. To maliciously affect a read request in a PASIS system, an intruder must compromise the *m* storage nodes serving the read request (assuming a share-verification scheme is in place). In a conventional system, an intruder compromising a primary server can cause it to return arbitrary values to read requests.

Required storage is the extra storage space a system needs beyond a single-copy baseline. In a PASIS system, the storage required depends on the threshold scheme being used. For example, secret sharing requires the same storage as replication, whereas information dispersal requires less storage than replication.

Latency is the delay a system experiences when it serves a request. In a conventional system, a client processes a read or write request by exchanging messages with a single server. In a PASIS system, messages must be exchanged with multiple storage nodes, which can significantly impact performance. However, some threshold schemes require that *m* servers each provide S/*m* of a dispersed object's S bytes. For large objects, network and client bandwidth limitations can potentially hide the overhead of contacting *m* servers.

2.5. PASIS architecture performance trade-offs

Threshold schemes can increase an information storage system's confidentiality, availability, and integrity. However, such schemes present trade-offs among information confidentiality, information availability, and storage requirements:

- As *n* increases, information availability increases (it's more probable that *m* shares are available), but the storage required for the information increases (more shares are stored) and confidentiality decreases (there are more shares to steal).
- As *m* increases, the storage required for the information decreases (a share's size is proportional to $1/(1+m-p)$), but so does its availability (more shares are required to reconstruct the original object). Also, as *m* increases, each share contains less information; this may increase the number of shares that must be captured before an intruder can reconstruct a useful portion of the original object.

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

- As *p* increases, the information system's confidentiality increases, but the storage space required for the dispersed information also increases.

With this flexibility, selecting the most appropriate threshold scheme for a given environment is not trivial.

Clients can also make trade-offs in terms of how they interact with storage nodes that hold shares. Even though only *m* shares are required to reconstruct an object, a client can over-request shares. That is, a client can send read requests to between *m* and *n* storage nodes. By over-requesting, the client reduces the risk of a data storage node being unavailable or slow.

2.6. Automatic trade-off selection

For the PASIS architecture to be as effective as possible, it must make the full flexibility of threshold schemes available to clients. We believe this option requires automated selection of appropriate threshold schemes on a per-object basis. This selection should combine object characteristics and observations about the current system environment. For example, a PASIS client could use short secret sharing to store an object larger than a particular size, and conventional secret sharing to store smaller objects. The size that determines which threshold scheme to use could be a function of the object type, current system performance, or both.

As another example, an object marked as archival—for which availability and integrity are the most important storage characteristics—should use an extra-large *n*. For read/write objects, increased write overhead makes large *n* values less desirable. Moreover, if the archival object is also marked as public—such as a Web page—the client should ignore confidentiality guarantees when selecting the threshold scheme.

System performance observations can also be used to dynamically improve per-request performance. For example, clients can request shares from the *m* storage nodes that have responded most quickly to their recent requests. Storage nodes can also help clients make these decisions by providing load information or by asking them to look elsewhere when long response times are expected.

As mentioned earlier, clients can use over-requesting to improve performance. For example, in an ad-hoc network with poor message-delivery guarantees, a PASIS client could notice the request loss rate and increase the number of shares requested on a read. On the other hand, in a very busy environment, the excess load on storage nodes inherent to over-requesting can reduce performance.

2.7. PASIS summary

The PASIS architecture combines proven technologies (threshold schemes and decentralized storage) to construct storage systems whose availability, confidentiality, and integrity policies can survive component failures and malicious attacks. The main challenges in deploying these systems relate to their engineering. Specifically, we need implementation techniques to help these systems achieve performance and manageability competitive with today's non-survivable storage systems. Our continuing research addresses this requirement by aggressively exploiting the flexibility offered by general threshold schemes within the PASIS architecture.

3. Self-securing storage

Secure storage servers have great difficulty coping with undesirable requests from legitimate user accounts. These requests can originate from malicious users, rogue programs run by unsuspecting users (e.g., e-mail viruses), or intruders exploiting compromised user accounts. Unfortunately, since there is no way for the storage system to distinguish an intruder from the real user, the system must treat them identically. Thus, the intruder can read or write anything to which the real user has access. In particular, they can modify or delete any or all of the accessible 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, manipulate file modification times, 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. Restoration is difficult because diagnosis is difficult and because user-convenience is an important issue. In addition, 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 client system is

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

vulnerable, including the raw storage. Rather than acting as slaves to client OSes, self-securing storage nodes 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 user-controlled versioning (e.g., Elephant [23][24], Cedar [13], or VMS [20]) is that client-side intruders can no longer bypass the versioning software by compromising complex OSes or their poorly-protected user accounts. Instead, intruders must compromise enough storage nodes to defeat the thresholding mechanisms described in the previous section.

This section describes the problems of client-side intrusion diagnosis and recovery, how self-securing storage addresses them, design challenges, and feasibility. Details of our prototype implementation, performance evaluation, and related work can be found in [26].

3.1. 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.

3.1.1. 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 [9], 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 [15]. 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.

3.1.2. 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

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

process often takes a considerable amount of time–time during which users are denied service.

3.2. Self-securing storage design

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.

3.2.1. Enabling intrusion survival. Overall, 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.

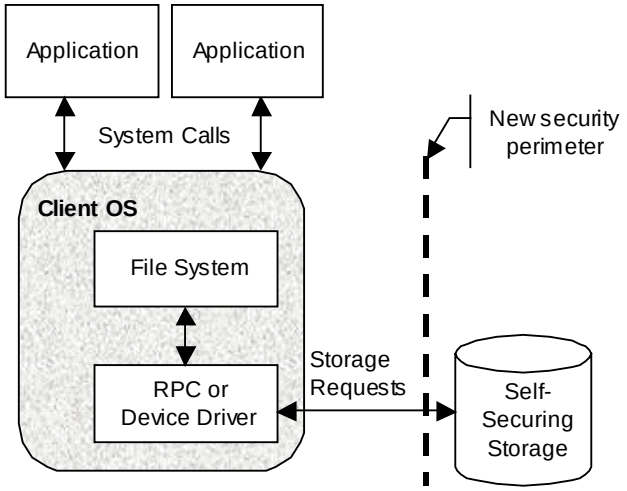

Figure 4. New security perimeter. Because the storage node executes distinct software on distinct hardware, compromising the client operating system does not circumvent the new security perimeter. Also note that the new perimeter complements (rather than replaces) any existing operating system and firewall perimeters.

3.2.2. 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 single-function device, the task of making it secure is much easier; compromising its firmware is analogous to breaking into an IDE or SCSI disk. Figure 4 illustrates this new security perimeter.

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

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

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.

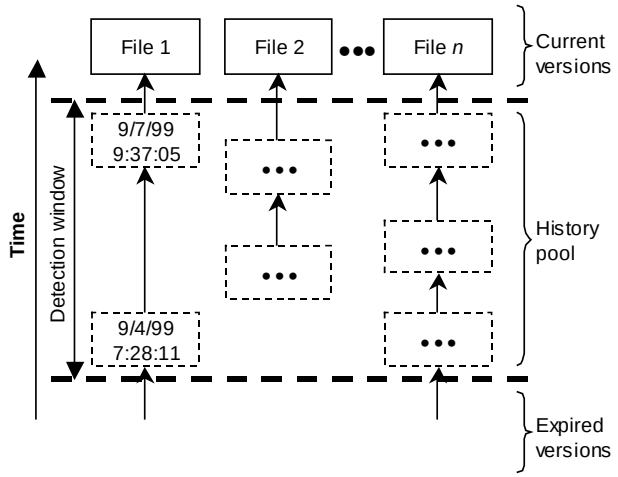

Figure 5. File versions in self-securing storage node. The most current versions are shown at the top. The history pool consists of the space holding all retained historical versions. The detection window, shown as the time between the two dashed lines, represents the guaranteed window of time during which the full history of changes is available to the administrator. Beyond that point, some or all of the versions may have expired and been removed from the history pool.

3.2.3. History pool management. Figure 5 illustrates the in-device versioning done behind the new security perimeter. 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:

- Providing a reasonable detection window in exceptionally busy systems.
- 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 3.3 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

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

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.

3.2.4. 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 electively 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.

3.2.5. 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.

3.2.6. 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 [23], 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.

3.3. Feasibility of self-securing storage

There are two basic feasibility concerns with the self-securing storage concept: performance and capacity. To address the former, we have constructed a prototype self-securing NFS server [26]. Experiments with this prototype show that the security and data survivability

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE

benefits of self-securing storage can be realized with reasonable performance. Specifically, its performance is comparable to FreeBSD's NFS for both micro-benchmarks and application benchmarks. Further, detailed analysis shows that the fundamental costs associated with self-securing storage degrade performance by less than 13% relative to similar systems that provide no data protection guarantees.

To evaluate the capacity requirements of self-securing storage, three recent workload studies were examined. Figure 6 shows the results of approximations based on worst-case write behavior. Spasojevic and Satyanarayanan's AFS trace study [27] 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 [29], 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. [24] 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.

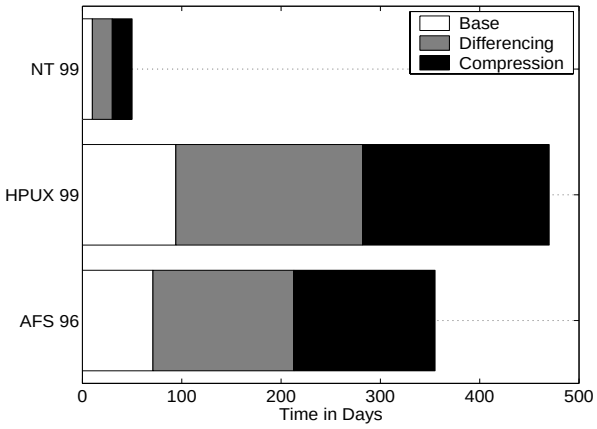

Figure 6. Projected detection window for three environments. This chart shows the expected detection window (in days) that could be provided by utilizing 10GB for data versioning. This conservative history pool would consume only 20% of a 50GB disk's total capacity. The base-line number represents the 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.

Much work has been done in evaluating the efficiency of differencing and compression [4][5][6]. To briefly explore the potential benefits for self-securing storage, 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 [18][19]. 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.

3.4. Self-securing storage summary

Self-securing storage nodes can ensure data and audit log survival in the presence of successful client-side intrusions, user and client impersonation, and even compromised client operating systems. Experiments (in [26]) with a prototype show that self-securing storage nodes can achieve performance that is comparable to existing storage servers. 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. Future work will focus on developing diagnosis and recovery tools that utilize the extensive information provided by self-securing storage.

4. Acknowledgements

This work is partially funded by DARPA/ATO's Organically Assured and Survivable Information Systems (OASIS) program (Air Force contract number F30602-99-2-0539-AFRL) and by the National Science Foundation via CMU's Data Storage Systems Center. We thank the members and companies of the Parallel Data Consortium (including EMC, HP, Hitachi, IBM, Infineon, Intel, LSI Logic, Lucent, MTI, Network Appliance, Novell, PANASAS, Platys, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. We also thank IBM Corporation for supporting our research efforts. Craig Soules is supported by a USENIX Fellowship.

References

[1] R. Anderson, "The Eternity Service," Proceedings PRAGOCRYPT 96, CTU Publishing House, Prague, 1996.

[2] T. Anderson, et al., “Serverless Network File Systems,” ACM Transactions on Computer Systems, February 1996, pp. 41-79.

[3] G. Blakley, “Safeguarding Cryptographic Keys,” Proceedings of the National Computer Conference of American Federation of Information Processing Societies, Montvale, N.J., 1979, pp. 313-317.

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

[5] M. Burrows and D. Wheeler, “A block-sorting lossless compression algorithm,” Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 10 May 1994.

[6] M. Burrows, C. Jerian, B. Lampson, and T. Mann, “On-line data compression in a log structured file system,” Architectural Support for Programming Languages and Operating Systems, October 1992.

[7] M. Castro, and B. Liskov, “Practical Byzantine Fault Tolerance,” *Operating Systems Review*, ACM Press, New York, 1999, pp. 173-186.

[8] A. De Santis, and B. Masucci, ”Multiple ramp schemes,” IEEE Transactions on Information Theory, July 1999, pp. 1720-1728.

[9] D. Denning, “An intrusion-detection model,” IEEE Transactions on Software Engineering, February 1987, pp. 222-232.

[10] Y. Deswarte, L. Blain, and J. Fabre, “Intrusion Tolerance in Distributed Computing Systems,” IEEE Symposium on Security and Privacy, 1991, pp. 110-121.

[11] G. A. Gibson, D. F. Nagle, W. Courtright II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka, “NASD scalable storage systems,” USENIX 99, Monterey, Ca., June 1999.

[12] A. Goldberg, and P. Yianilos, “Prototype implementation of archival Intermemory,” in Proceedings of IEEE ADL, April 1998, pp. 147-156.

[13] R. Hagman, “Reimplementing the Cedar file system using logging and group commit,” ACM Symposium on Operating System Principles, November 1987.

[14] A. Iyengar, R. Cahn, J. Garay, and C. Jutla, “Design and Implementation of a Secure Distributed Data Repository,” in Proceedings of the 14th IFIP International Information Security Conference (SEC '98), September 1998.

[15] G. Kim and E. Spafford, “The design and implementation of Tripwire: a file system integrity checker,” Conference on Computer and Communications Security, November, 1994, pp. 18-29.

[16] H. Krawczyk, “Secret sharing made short”, Advances in Cryptology - CRYPTO '93,. 13th Annual International Cryptology Conference Proceedings, 1993, pp. 136-146.

[17] E. Lee, C. Thekkath, “Petal: Distributed Virtual Disks,” Proceedings of ACM ASPLOS, October 1996, pp. 84-92.

[18] J. MacDonald, *File System support for delta compression*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[19] J. MacDonald, P Hilfinger, A. Costello, R Wang, and T. Anderson, “Improving the performance of log-structured file systems with adaptive methods,” ACM Symposium on Operating Systems Principles, 1997.

[20] K. McKoy, *VMS file system internals*, Digital Press, 1990.

[21] M. Rabin, “Efficient dispersal of information for security, load balancing, and fault tolerance,” Journal of the Association for Computing Machinery, 1989, pp. 335-348.

[22] M. Reiter, “Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart,” in Proceedings of the 2nd ACM Conference on Computer and Communication Security, November 1994, pp. 68-80.

[23] D. Santry, M. Feeley, and N. Hutchinson, “Elephant the file system that never forgets,” Hot Topics in Operating Systems, IEEE CS, 1999.

[24] D. Santry, M. Feeley, N. Hutchinson, R. Carton, J. Ofir, and A. Veitch, “Deciding when to forget in the Elephant file system, “ACM Symposium on Operating System Principles, 1999.

[25] A. Shamir, “How to Share a secret,” Communications of the ACM, 1979, pp. 612-613.

[26] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, G. Ganger, “Design and Implementation of a Self-Securing Storage Device”, Symposium on Operating Systems Design and Implementation, October 2000, pp. 165-179.

[27] M. Spasojevic and M. Satyanarayanan, “An empirical study of wide-area distributed file system,” ACM Transactions on Computer Systems, May 1996, pp. 200-222.

[28] M. Tompa, and H. Woll, “How to share a secret with cheaters,” Journal of Cryptology, 1988, pp. 133-138.

[29] W. Vogels, “File system usage in Windows NT 4.0,” ACM Symposium on Operating System Principles, 1999.

[30] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, Pradeep K. Khosla, "Survivable Information Storage Systems," IEEE Computer, August 2000, 61-68.

ISBN 0-7695-1212-7/01 $10.00 (C) 2001 IEEE