Storage-based Intrusion Detection: Watching storage activity for suspicious behavior

Adam G. Pennington, John D. Strunk, John Linwood Griffin,
Craig A.N. Soules, Garth R. Goodson, Gregory R. Ganger

Oct 2002

CMU-CS-02-179

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Storage-based intrusion detection allows storage systems to transparently watch for suspicious activity. Storage systems are well-positioned to spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and tampering with audit logs. Further, an intrusion detection system (IDS) embedded in a storage device continues to operate even after client systems are compromised. This paper describes a number of specific warning signs visible at the storage interface. It describes and evaluates a storage IDS, embedded in an NFS server, demonstrating both feasibility and efficiency of storage-based intrusion detection. In particular, both the performance overhead and memory required (40 KB for a reasonable set of rules) are minimal. With small extensions, storage IDSs can also be embedded in block-based storage devices.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by DARPA/ITO's OASIS program, under Air Force contract number F30602-99-2-0539-AFRL. Craig Soules was supported by a USENIX Fellowship. Garth Goodson was supported by an IBM Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

Keywords: Intrusion detection, IDS, virus detection, computer security.

1 Introduction

Digital intrusions[1] are a fact of modern computing. Although new security technologies may make them more difficult and less frequent, intrusions will occur as long as software is imperfect and users are fallible. Further, misbehaving "insiders" are a growing problem [32, p.47]. As a result, intrusion detection is an important part of a managed computing environment.

An intrusion detection system (IDS) attempts to identify attacks on and successful breaches of security perimeters. Many digital IDSs have been developed over the years [3, 23, 28], with most falling into one of two categories: network-based and host-based. Network IDSs are usually embedded in sniffers or firewalls, scanning traffic to, from, and within a network environment for attack signatures and suspicious traffic [8, 26]. Host-based detection systems are fully or partially embedded within each endhost's OS, examining local information (such as system calls [13]) for signs of intrusion or suspicious behavior. Many environments employ multiple IDSs, each watching activity from its own vantage point.

The storage system is another interesting vantage point for intrusion detection. Several common intruder actions [11, p.218][31, pp.363–365] are quite visible at the storage interface. Examples include manipulating system utilities (e.g., to add backdoors or Trojan horses), tampering with audit log contents (e.g., to eliminate evidence), and resetting attributes (e.g., to hide changes). By design, a storage server sees all changes to stored data, allowing it to transparently watch for suspicious changes and issue alerts about the corresponding client systems. Also, like network-based IDSs, a storage IDS can be compromise-independent of host OSes, meaning that it cannot be disabled by an intruder that successfully gets past a host's OS-level protection.

This paper motivates and describes storage-based intrusion detection. We describe several kinds of suspicious behavior that can be spotted by a storage IDS, focusing on specific rules whose violation often indicates malicious behavior. We discuss design issues faced in realizing storage-based intrusion detection, including administration and possible responses. We demonstrate feasibility with a prototype storage IDS.

A storage IDS could be embedded in many kinds of storage systems. It would require extra processing power and memory space, which should be feasible in file servers, disk array controllers, and perhaps augmented disk drives. Most detection rules will also require FS-level understanding of the stored data. Such understanding exists trivially for a file server, but it must be explicitly provided to block-based storage devices.

As a concrete example, we have augmented an NFS server with a storage IDS that supports rule-based detection of suspicious modifications. This storage IDS allows the server to warn an administrator of unexpected changes to important system files and binaries, based on a rule-set very similar to Tripwire [19]. Additional rules detect system log tampering and suspicious attribute modification. An administrative interface supplies the detection rules, which are checked during the processing of each NFS request. When a detection rule fires, the server sends the administrator an alert containing the full pathname of the modified file, the violated rule, and the offending NFS operation. Experiments show that the runtime cost of such intrusion detection is minimal in the common case of no intrusions detected. Further analysis indicates that little memory capacity is needed for reasonable rulesets (e.g., only 40 KB for an example containing 391 rules).

[1] We use "intrusion" as a catch-all term for any access to and use of computer systems not authorized by their owners, including misuse by insiders, break-ins resulting in interactive use by intruders, and self-propagating viruses and worms.

The paper also discusses the functionality required for storage-based intrusion detection in devices exporting a block-based interface. To implement the same rules as the extended NFS server, such a device must be able to parse and traverse the on-disk metadata structures of the file system it holds. For example, knowing whether `/usr/sbin/sshd` has changed on disk requires knowing not only whether the corresponding data blocks have changed, but also whether the inode still points to the same blocks and whether the name still translates to the same inode. Achieving this degree of FS-specific information is quite feasible, assuming that the device firmware is extended with a small amount of additional software. For two popular file systems (Linux's ext2fs and FreeBSD's FFS), the additional functionality required is small (under 200 lines of C code for each), simple (under 3 days of programming effort each), and stable (about 5 years between changes to on-disk structures).

The remainder of this paper is organized as follows. Section 2 overviews storage-based intrusion detection. Section 3 discusses storage IDS design issues. Section 4 describes a prototype storage IDS embedded in an NFS server. Section 5 uses this prototype to evaluate storage-based intrusion detection. Section 6 discusses extensions needed for storage IDSs in block-based storage components. Section 7 discusses related work. Section 8 summarizes this paper's contributions.

2 Storage-based Intrusion Detection

Storage-based intrusion detection consists of having storage servers examine the requests they service for suspicious client behavior. Although the world view that a storage server sees is incomplete, two features combine to make it a well-positioned platform for enhancing intrusion detection efforts. First, since storage servers are independent of client OSes, they can continue to look for intrusions after the initial compromise, whereas a host-based IDS can be disabled by the intruder. Second, since most computer systems rely heavily on persistent storage in their operation, many intruder actions will cause storage activity that can be captured and analyzed.

This section expands on these two features, discusses some options for on-line response, and identifies limitations of storage-based intrusion detection.

2.1 Compromise independence

A storage IDS will continue watching for suspicious activity even when clients' OSes are compromised. It capitalizes on the fact that storage servers (whether file servers, disk array controllers, or even IDE disks) run separate software on separate hardware, as illustrated in Figure 1. This fact enables server-embedded security functionality that cannot be disabled by any software (even the OS kernel) running on client systems. Further, since they often have fewer network interfaces (e.g., RPC+SNMP+HTTP or even just SCSI) and no local users, compromising a storage server should be more difficult than compromising a client system. Of course, such servers have a limited view of system activity, so they cannot distinguish legitimate users from clever impostors. But, from behind the physical storage interface, a storage IDS can spot many common intruder actions and alert administrators.

Administrators must be able to communicate with the storage IDS, both to configure it and to receive alerts. This administrative channel must also be compromise-independent of client systems, meaning that no user (even root) and no software (even the OS kernel) on a client system

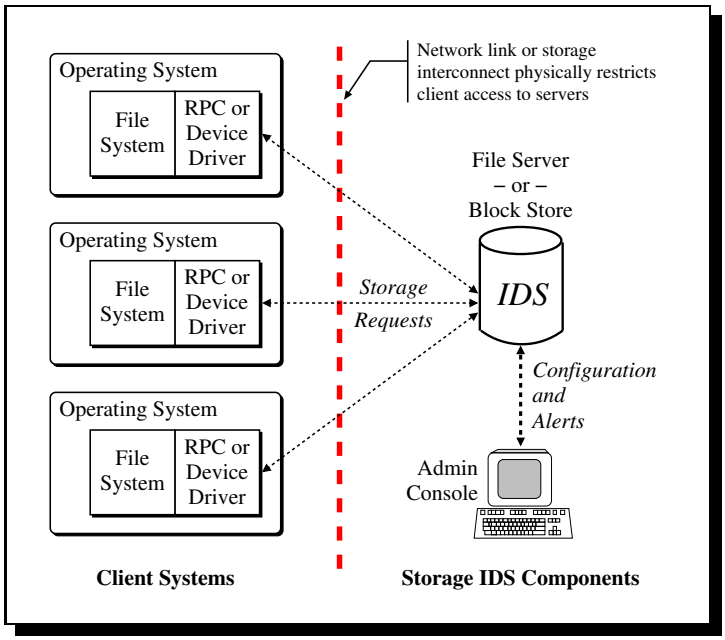

Figure 1: **The compromise independence of a storage IDS.** The storage interface provides a physical boundary behind which a storage server can observe the requests it is asked to service. Note that this same picture works for block protocols, such as SCSI or IDE/ATA, and distributed file system protocols, such as NFS or CIFS. Thus, storage-based intrusion detection could be realized within many storage servers, including file servers, disk array controllers, and even disk drives.

can have administrative privileges for the storage IDS. Section 3 discusses deployment options, including physical consoles and cryptographic channels from a dedicated administrative system.

All of the warning signs discussed in this paper could also be spotted from within hosts' OS software, but host-based IDSs do not enjoy the compromise independence of storage IDSs. A host-based IDS is vulnerable to being disabled or bypassed by intruders that compromise the OS kernel. Another interesting place for a storage IDS is the virtual disk module of a virtual machine monitor [36]; such deployment would enjoy compromise independence from the OSes running in its virtual machines [7].

2.2 Warning signs for storage IDSs

Successful intruders often modify stored data. For instance, they may remove traces of the system penetration, install Trojan horse programs to capture passwords, install a back door for future re-entry, or tamper with data files for the purpose of sabotage. These modifications will be visible to the storage system when they are made persistent. This section describes four categories of signals that a storage IDS can monitor: particular update operations, update patterns, structural changes to content, and actual content values.

2.2.1 Data/attribute modification

In managed computing environments, the simplest (and perhaps most effective) category of warning signs consists of any changes to files that administrators expect to remain unchanged except

for explicit upgrades. Examples of such files include system executables and scripts, configuration files, and system header files and libraries. Given the importance of such files and the infrequency of updates to them, every modification is a warning sign not to be ignored. A storage IDS can detect all such modifications on-the-fly, before the storage device processes each request, and issue an alert immediately. An administrator making legitimate updates can quickly dismiss the expected alert; unexpected alerts should be investigated. Alternately, the storage IDS could include an administrative function for informing the device of the forthcoming change in advance.

In current systems, modification detection is sometimes provided by a checksumming utility (e.g., Tripwire [19]) that periodically compares the current storage state against a reference database stored elsewhere. Storage-based intrusion detection improves on this current approach in four ways: (1) it allows immediate detection of changes to watched files; (2) it avoids the complexity of maintaining and protecting a reference database; (3) for local storage, it avoids relying on the client OS to do the checks, which a successful intruder could disable or bypass; and (4) it can notice short-term changes, made and then undone, which would not be noticed by a checksumming utility if they occurred between two periodic checks.

2.2.2 Update patterns

A second category of warning signs consists of suspicious access patterns, particularly updates. There are several concrete examples for which storage IDSs can usefully watch.

The clearest examples are client system audit logs. These audit logs are critical to both intrusion detection [10] and diagnosis [33], leading many intruders to scrub any evidence from them as a precaution. Any such manipulation will be obvious to a storage IDS that understands the well-defined update pattern of the specific audit log. For instance, audit log files are usually append-only, and they may be periodically "rotated." This rotation consists of renaming the current log file to an alternate name (e.g., `logfile` to `logfile.0`) and creating a new "current" log file. Any deviation in the update pattern of the current log file or any update to a previous log file is suspicious.

Another suspicious update pattern is timestamp reversal. Specifically, the data modification and attribute change times commonly kept for each file can be quite useful for post-intrusion diagnosis of what files were manipulated [12]. By manipulating the times stored in inodes (e.g., to set them back to their original values), an intruder can inhibit such diagnosis. Of course, care must be taken with IDS rules, since some programs (e.g., `tar`) legitimately set these times to old values. A possible rule would trigger an alert when someone sets the modification time back on a file that was not just created. This would exclude `tar`-style activity but would catch an intruder trying to obfuscate a modified file. Of course, the intruder could now delete the file, create a new one, set the date back, and hide from the storage IDS—a more complex rule could catch this, but such escalation is the nature of intrusion detection.

Detection of storage denial-of-service (DoS) attacks also falls into the category of suspicious access patterns. For example, an attacker can disable specific services or entire systems by allocating all or most of the free space. A similar effect can be achieved by allocating inodes or other metadata structures. A storage IDS can watch for such exhaustion, which may be deliberate, accidental, or coincidental (e.g., a user just downloaded 10 GB of trace files). When the system reaches predetermined thresholds of unallocated resources and allocation rate, warning the administrator is appropriate even in non-intrusion situations—attention is likely to be necessary soon. A

storage IDS could similarly warn the administrator when server load exceeds a threshold for too long, which may be a DoS attack (or just an indication that the server needs to be upgraded).

Although specific rules can spot expected intruder actions, more general rules may allow larger classes of suspicious activity to be noticed. For example, some attribute modifications, like enabling "set UID" bits or reducing the permissions needed for access, may indicate foul play. Additionally, many applications access storage in a regular manner. As two examples: word processors often use temporary and backup files in specific ways, and UNIX password management involves a pair of inter-related files (`/etc/passwd` and `/etc/shadow`). The corresponding access patterns seen at the storage device will be a reflection of the application's requests. This presents an opportunity for anomaly detection based on how a given file is normally accessed. This could be done in a manner similar to learning common patterns of system calls [13] or starting with rules regarding the expected behavior of individual applications [20]. Deviation from the expected pattern could indicate an intruder attempting to subvert the normal method of accessing a given file. Of course, the downside is an increase (likely substantial) in the number of false alarms. Nonetheless, anomaly detection within storage access patterns is an interesting topic for future research.

2.2.3 Content integrity

A third category of warning signs consists of changes that violate internal consistency rules of specific files. This category goes beyond the update pattern by understanding the application-specific semantics of particularly important stored data. Of course, to verify content integrity, the device must understand the format of a file. Further, while simple formats may be verified in the context of the write operation, file formats may be arbitrarily complex and verification may require access to additional data blocks (other than those currently being written). This creates a performance vs. security trade-off made by deciding which files to verify and how often to verify them. In practice, there are likely to be few critical files for which content integrity verification is utilized.

As a concrete example, consider a UNIX system password file (`/etc/passwd`), which consists of a set of well-defined records. Records are delimited by a line-break, and each record consists of seven colon-separated fields. Further, each of the fields has a specific meaning, some of which are expected to conform to rules of practice. For example, the seventh field specifies the "shell" program to be launched when a user logs in, and (in Linux) the file `/etc/shells` lists the legal options. During the "Capture the Flag" information warfare game at the 2002 DEF CON conference [9], one tactic used was to change the root shell on compromised systems to `/sbin/halt`; once a targeted system's administrator noted the intrusion and rebooted the machine (the common initial reaction), considerable down-time and administrative effort was needed to restore the system to operation. A storage IDS can monitor changes to `/etc/passwd` and verify that they conform to a set of basic integrity rules: 7-field records, non-empty password field, legal default shell, legal home directory, non-overlapping user IDs, etc. The attack described above, among others, could be caught immediately.

A larger example is offered by the write-ahead logs used in many database and file systems. Planned changes are committed to such logs before being made to the actual data or metadata, allowing post-crash recovery to a consistent state. Such logs can also be used for intrusion diagnosis and recovery [1, 2]. Intruders can bypass and manipulate these intrusion survival mechanisms by manipulating the data directly, without allowing the changes to appear in the log. A powerful

storage IDS could ensure that all changes are properly reflected in the log and that no fake changes appear in the log, warning the administrator whenever these system invariants are violated.

2.2.4 Suspicious content

A fourth category of warning sign is the appearance of suspicious content. The most obvious suspicious content is a known virus, detectable via its signature. Several high-end storage servers (e.g., from EMC [24] and Network Appliance [27]) now include support for internal virus scanning. By executing the scans within the storage server, viruses cannot disable the scanners even after infecting clients.

Two other examples of suspicious content are large numbers of "hidden" files or empty files. Hidden files have names that are not displayed by normal directory listing interfaces [11, p.217], and their use may indicate that an intruder is using the system as a storage repository, perhaps for illicit or pirated content. A large number of empty files or directories may indicate an attempt to exploit a race condition [4, 29] by inducing a time-consuming directory listing, search, or removal.

2.3 Responding to intrusions

In general, a storage IDS should operate in a way that will not interfere with a valid, running system. Since a detected "intruder action" may actually be legitimate user activity (i.e., a false alarm), our default response is simply to send an alert to the administrative system or the designated alert log file. There are, however, more active responses that a storage IDS could trigger upon detecting suspicious activity. When choosing the proper response, the administrator must weigh the benefits of an active response against the inconvenience and potential damage caused by false alarms.

One reasonable active response is to slow down the suspected intruder's storage accesses. For example, a storage device could wait until the alert is acknowledged before completing the suspicious request. It could also artificially increase request latencies for a client or user that is suspected of foul play. Doing so would provide increased time for a more thorough response, and, while it will cause some annoyance in false alarm situations, it is unlikely to cause damage. The device could even deny a request entirely if it violates one of the rules, although this response to a false alarm could cause damage and/or application failure.

Liu, et al. proposed a more radical response to detected intrusions: isolating intruders, via versioning, at the file system level [22]. To do so, the file system forks the version trees to sandbox suspicious users until the administrator verifies the legitimacy of their actions. Unfortunately, such forking is likely to interfere with system operation, unless the intrusion detection mechanism yields no false alarms. Specifically, since suspected users modify different versions of files from regular users, the system faces a difficult reintegration [21, 39] problem, should the updates be judged legitimate. Still, it is interesting to consider embedding this approach, together with a storage IDS, into storage systems for particularly sensitive environments.

A less intrusive storage-embedded response is to start versioning all data and auditing all storage requests when an intrusion is detected. Doing so provides the administrator with significant information for post-intrusion diagnosis and recovery [35]. Of course, some intrusion-related information will likely be lost unless the intrusion is detected immediately, which is why Strunk

et al. argue for always doing these things (just in case). Still, IDS-triggered employment of this functionality presents a useful trade-off point.

2.4 Limitations

Although storage-based intrusion detection contributes to security efforts, it is not a silver bullet.

Like any IDS, a storage IDS will produce some false alarms. With very specific rules, such as "watch these 100 files for any modification," false alarms should be infrequent; they will occur only when there are legitimate changes to a watched file, which should be easily verified if updates involve a careful procedure. The issue of false alarms grows progressively more problematic as the rules get less exact (e.g., the time reversal or resource exhaustion examples). The far end of the spectrum from specific rules is general anomaly detection.

Also like any IDS, a storage IDS will fail to spot some intrusions. Fundamentally, a storage IDS cannot notice intrusions whose actions do not cause odd storage behavior. For example, the intrusion may be a worm program that communicates over the network but changes no files. Or, an intruder may manipulate storage in unwatched ways. Using network-based and host-based IDSs together with a storage IDS can increase the odds of spotting various forms of intrusion.

Intrusion detection, as an aspect of information warfare, is by nature a "game" of escalation. As soon as one side takes away an avenue of attack, the other must start looking for the next. Since storage-based intrusion detection easily sees several common intruder activities, crafty intruders will change tactics. For example, an intruder can do anything she wants in the host's memory, so long as nothing fishy propagates to storage. A reboot will reset the system, which argues for proactive restart [6, 17, 41]. To counter this, some attackers will identify ways of having their changes re-established automatically after a reboot, such as by manipulating the various boot-time (e.g., `rc.local` in UNIX-like systems) or periodic (e.g., `cron` in UNIX-like systems) programs. In turn, security administrators will learn to construct rules that prevent or spot such changes. And, the "game" continues.

As a practical consideration, storage IDSs embedded within individual components of decentralized storage systems are unlikely to be effective. For example, a disk array controller is a fine place for storage-based intrusion detection, but individual disks behind software striping are not; each of the disks has only part of the file system's state, making it difficult to check non-trivial rules without adding new inter-device communication paths.

Finally, storage-based intrusion detection is not free. Checking rules comes with some cost in processing and memory resources, and more rules require more resources. In configuring a storage IDS, one must balance detection efforts with performance costs for the particular operating environment. Section 5 quantifies some of these costs.

3 Design of a Storage IDS

To be useful in practice, a storage IDS must simultaneously achieve several goals. It must support a useful set of detection rules, while also being easy for human administrators to understand and configure. It must be efficient, minimizing both added delay and added resource requirements; some user communities still accept security measures only when they are "free." Additionally, it should be invisible to users at least until an intrusion detection rule is matched.

This section describes four aspects of storage IDS design: specifying detection rules, administering a storage IDS securely, verifying detection rules, and responding to suspicious activity.

3.1 Specifying detection rules

Specifying rules for an IDS is a tedious, error prone activity. The tools an administrator uses to write and manipulate those rules should be as simple and straightforward as possible. Each of the categories of suspicious activity that a storage IDS can look for will likely need a different format for rule specification.

The rule format used by Tripwire seems to work well for specifying rules concerned with data and attribute modification. This format allows an administrator to specify the pathname of a file and a list of properties that should be monitored for that file. The set of watchable properties are codified, and they include most file attributes. This rule language works well, because it allows the administrator to manipulate a representation that he can understand (pathnames and files), and the list of attributes that can be watched is small and well-defined.

When configuring the IDS to look for suspicious content, the methods used by host-based virus scanners are applicable. That is, "rules" can be specified as data signatures for which to look.

Less experience exists for the other two categories of warning signs: update patterns and content integrity. For update patterns, one could use lists of legal or illegal operations. For content integrity, pattern matching languages like `yacc` provide guidance. Approaches to specifying detection rules for these categories of warning signs is an area of future work.

3.2 Secure administration

The security administrator must have a secure interface to the storage IDS. This interface is needed for the administrator to configure detection rules and to receive alerts. The interface must prevent client systems from forging or blocking administrative requests, since this could allow a crafty intruder to sneak around the IDS by disarming it. At a minimum, it must be tamper-evident. Otherwise, intruders could stop rule updates or prevent alerts from reaching the administrator. To maintain compromise independence, it must be the case that obtaining "superuser" or even kernel privileges on a client system is insufficient to gain administrative access to the storage device.

Two promising architectures exist for such administration: one based on physical access and one based on cryptography. For environments where the administrator has physical access to the device, a local administration terminal that allows the administrator to set detection rules and receive the corresponding alert messages satisfies the above goals.

In environments where physical access to the device is not practical, cryptography can be used to secure communications. In this scenario, the storage device acts as an endpoint for a cryptographic channel to the administrative system. The device must maintain keys and perform the necessary cryptographic functions to detect modified messages, lost messages, and blocked channels. Architectures for such trust models in storage systems exist [15]. This type of infrastructure is already common for administration of other network-attached security components, such as firewalls or network intrusion detection systems. For direct-attached storage devices, cryptographic channels can be used to tunnel administrative requests and alerts through the OS of the client system, as illustrated in Figure 2. Such tunneling simply treats the client OS as an untrusted network component.

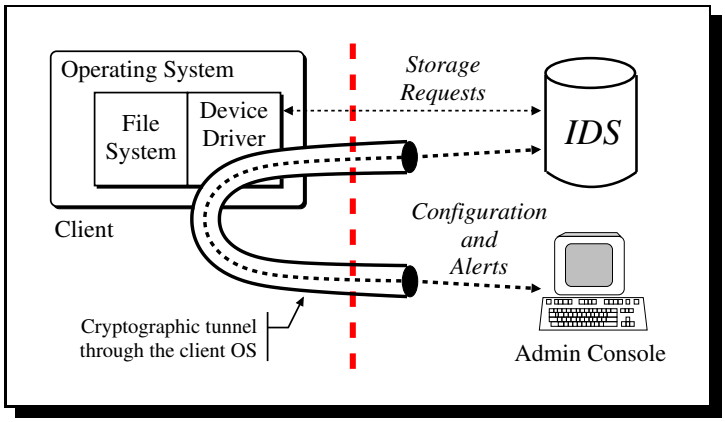

Figure 2: **Tunneling administrative commands through client systems.** For storage devices attached directly to client systems, a cryptographic tunnel can allow the administrator to securely manage a storage IDS. This tunnel uses the untrusted client OS to transport administrative commands and alerts.

For small numbers of dedicated servers in a machine room, either approach is feasible. For large numbers of storage devices or components operating in physically insecure environments, cryptography is the only viable solution.

3.3 Checking the detection rules

Specified detection rules need to be checked completely and efficiently. This can be non-trivial, because rules generally apply to full pathnames rather than inodes. Additional complications arise because rules can watch for files that do not yet exist.

For simple operations that act on individual files (e.g., READ and WRITE), rule verification is straightforward. The device need only check that the rules pertaining to that specific file are not violated (a simple flag comparison). For operations that affect the file system's namespace, verification is more complicated. For example, a rename of a directory tree may impact a large number of individual files, any of which could have IDS rules that must be checked. Renaming a directory requires examining all files and directories that are children of the one being renamed.

In the case of rules pertaining to files that do not currently exist, this list of rules must be consulted when operations change the namespace. For example, the administrator may want to watch for the existence of a file named `/a/b/c` even if the directory named `/a` does not yet exist. However, a single file system operation (e.g., `mv /z /a`) could cause the watched file to suddenly exist, given the appropriate structure for `z`'s directory tree.

3.4 Responding to rule violations

Although Section 2.3 discusses active responses, most of this paper assumes that the only storage IDS reaction to a violated detection rule is to send an alert to an administrative system. The alert message should contain enough information for an administrator to understand and investigate the potential intrusion. Necessary information includes some reference to the file(s) involved, the time of the event, and the action being performed. Additionally, an administrator can be informed of the action's attributes (such as the actual data written into a file) or of the client's identity.

While we discuss the alert notification mechanism as though it is destined for the administrator directly, it could be sent to an audit log or to automated security tools for further evidence gathering before it is presented to the administrator. Additionally, if the receiving system is involved in normal administrative activities such as upgrades, it could correlate the generated alert with an in-progress upgrade and not forward it to the administrator.

4 Storage-based intrusion detection in an NFS server

To explore the concepts and feasibility of storage-based intrusion detection, we implemented a storage IDS in an NFS server. Unmodified client systems access the server using the standard NFS version 2 protocol [37], while storage-based intrusion detection occurs transparently. This section describes how the storage IDS handles detection rule specification, the structures and algorithms for checking rules, and alert generation.

The base NFS server is called S4, and its implementation is described and evaluated elsewhere [35]. It internally performs file versioning and request auditing, using a log-structured file system [30], but these features are not relevant here. For our purposes, it is a convenient NFS file server with performance comparable to the Linux and FreeBSD NFS servers. Secure administration is performed via the server's console, using the physical access control approach.

4.1 Specifying detection rules

Our prototype storage IDS is capable of watching for data and metadata changes to files. The administrator specifies a list of Tripwire-like rules to configure the detection system. Each administrator-supplied rule is of the form: {*pathname, attribute-list*}—designating which attributes to monitor for a particular file. The list of attributes that can be watched is shown in Table 1. In addition, an administrator can choose to enable either of two additional rules: one that matches on any operation that rolls-back a file's modification time, and one that matches on any operation that creates a "hidden" file (e.g., a file beginning with "."). Both rules apply to all parts of the directory hierarchy, and they are currently pre-installed and specified as simply ON or OFF.

Rules are communicated to the server through the use of an administrative RPC. This RPC interface has two commands (see Table 2). The `setRule()` RPC gives the IDS two values: the path of the file to be watched, and a set of flags describing the specific rules for that file. Rules are removed through the same mechanism, specifying the path and an empty rule set. We currently use a simple user-space utility to iterate over an ASCII configuration file and issue the appropriate `setRule()` RPCs.

4.2 Checking the detection rules

This subsection describes the core of the storage IDS. It discusses how rules are stored and subsequently checked during operation.

Metadata	
• inode modification time	• data modification time
• access time	• file permissions
• link count	• device number
• file owner	• inode number
• file type	• file owner group
• file size	
Data	
• any modification	• append only

Table 1: **Attribute list.** Rules can be established to watch these attributes in real-time on a file-by-file basis.

Command	Purpose	Direction
`setRule(path, rules)`	Changes the watched characteristics of a file. This command is used to both set and delete rules.	admin⇒server
`listRules()`	Retrieves the server's rule table as a list of {`pathname`, `watch`} records.	admin⇒server
`alert(path, rules, operation)`	Delivers a warning of a rule violation to the administrator.	server⇒admin

Table 2: **Administrative commands for our storage IDS.** This table lists the small set of administrative commands needed for an administrative console to configure and manage the storage IDS. The first two are sent by the console, and the third is sent by the storage IDS. The pathname refers to a file relative to the root of an exported file system. The *rules* are a description of what to check for, which can be any of the changes described in Table 1. The *operation* is the NFS operation that caused the rule violation.

4.2.1 Data structures

Three new structures allow the storage IDS to efficiently support the detection rules: the reverse lookup table, the inode watch flags, and the non-existent names table.

Reverse lookup table: The reverse lookup table lists the detection rules that apply to files and directories in the system. It serves two functions. First, it tracks detection rules that the server is currently enforcing. Second, it assists with translation of an inode number to a full pathname. The alert generation mechanism uses the latter to provide the administrator with file names instead of inode numbers, without resorting to a brute-force search of the namespace.

The reverse lookup table is populated via the `setRule()` RPC. Each rule's full pathname is broken into its component names, which are stored in distinct rows of the table. For each component, the table records four fields: *inode-number*, *directory-inode-number*, *name*, and *rules*. The first three indicate the *inode-number* corresponding to a given *name* within a particular directory (identified by its *directory-inode-number*). The *rules* associated with this *name* are a bitmask of the attributes and patterns to watch. Since a particular inode number can have more than one name, multiple entries for each inode may exist. A given inode number can be translated to a full

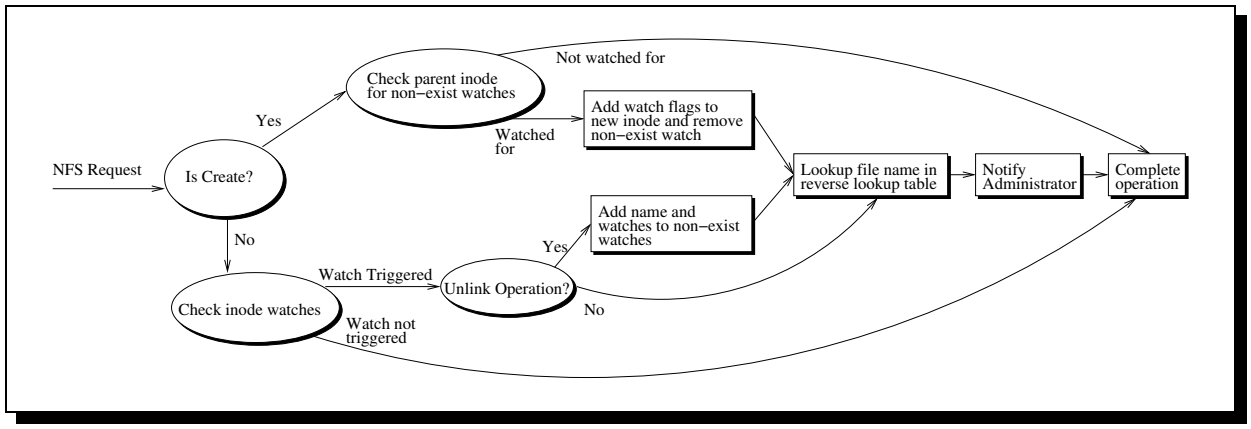

Figure 3: **Flowchart of our storage IDS.** Few structures and decision points are needed. In the common case (no rules for the file), only one inode's `watchflags` field is checked. The picture does not show RENAME operations here due to their complexity.

pathname by looking up its lowest-level name and recursively looking up the name of the corresponding directory inode number. The search ends with the known inode number of the root directory. All names for an inode can be found by following all paths given by the lookup of the inode number.

Inode `watchflags` field: During the `setRule()` RPC, in addition to populating the reverse lookup table, a rule mask of 16 bits is computed and stored in the `watchflags` field of the watched file's inode. Since multiple pathnames may refer to the same inode, there may be more than one rule for a given file, and the mask contains the union. The inode `watchflags` field is a performance enhancement designed to co-locate the rules governing a file with that file's metadata. This field is not necessary for correctness since the pertinent data could be read from the reverse lookup table. However, it allows efficient verification of detection rules during the processing of an NFS request. Since the inode is read as part of any file access, rule verification becomes a simple mask comparison.

Non-existent names table: The non-existent names table lists rules for pathnames that do not currently exist. Each entry in the table is associated with the deepest-level (existing) directory within the pathname of the original rule. Each entry contains three fields: *directory-inode-number*, *remaining-path*, and *rules*. Indexed by *directory-inode-number*, an entry specifies the *remaining-path*[2] to the non-existent file for which the *rules* apply. When a file or directory is created or removed, the non-existent names table is consulted and updated, if necessary. For example, upon creation of a file for which a detection rule exists, the *rules* are checked and inserted in the `watchflags` field of the inode. Together, the reverse lookup table and the non-existent names table contain the entire set of IDS rules in effect.

4.2.2 Checking rules during NFS operations

We now describe the flow of rule checking, much of which is diagrammed in Figure 3, in two parts: changes to individual files and changes to the namespace.

[2]One artifact of the current implementation is a maximum of 255 bytes for the *remaining-path* field. As a result, the storage IDS cannot currently watch pathnames longer than this.

Checking rules on individual file operations: For each NFS operation that affects only a single file, a mask of rules that might be violated is computed. This mask is compared, bitwise, to the corresponding `watchflags` field in the file's inode. This comparison quickly determines whether any of the rules are triggered, causing alert generation.

Checking rules on namespace operations: Namespace operations can cause watched pathnames to appear or disappear, which will usually trigger an alert. For operations that create watched pathnames, the storage IDS moves rules from the non-existent names table to the reverse lookup table. Conversely, operations that delete watched pathnames cause rules to move between tables in the opposite direction.

When a name is created (via CREATE, MKDIR, LINK, or SYMLINK) the non-existent names table is checked. If there are rules for the new file, they are checked and placed in the `watchflags` field of the new inode. In addition, the corresponding rule is removed from the non-existent names table and is added to the reverse lookup table. During a MKDIR, any entries in the non-existent names table that include the new directory as the next step in their remaining path are replaced; the new entries are indexed by the new directory's inode number and have its name removed from their remaining path.

When a name is removed (via UNLINK or RMDIR), the `watchflags` field of the corresponding inode is checked for rules. Most such rules will trigger an alert, and an entry for them is also added to the non-existent names table. For RMDIR, the reverse of the actions for MKDIR are necessary. Any non-existent table entries parented on the removed directory must be modified. The removed directory's name is added to the beginning of each remaining path, and the directory inode number in the table is modified to be the directory's parent.

By far, the most complex namespace operation is a RENAME. For a RENAME of an individual file, modifying the rules is the same as a CREATE of the new name and a REMOVE of the old. When a directory is RENAMEd, its subtrees must be recursively checked for watched files. If any are found, and once appropriate alerts are generated, their rules and pathname up to the parent of the renamed directory are stored in the non-existent names table, and the `watchflags` field of the inode is cleared. Then, the non-existent names table must be checked (again recursively) for any rules that map into the directory's new name and its children; such rules are checked, added to the inode's `watchflags` field, and updated as for name creation.

4.3 Generating alerts

Alerts are generated and sent immediately when a detection rule is triggered. The alert consists of the original detection rule (pathname and attributes watched), the specific attributes that were affected, and the RPC operation that triggered the rule. To get the original rule information, the reverse lookup table is consulted. If a single RPC operation triggers multiple rules, one alert is sent for each.

5 Evaluation

This section evaluates the costs of our storage IDS in terms of performance impact and memory required—both costs are minimal.

Benchmark	Baseline	With IDS	Change
SSH untar	27.3 (0.02)	27.4 (0.02)	0.03%
SSH config.	42.6 (0.68)	43.2 (0.37)	1.3%
SSH build	85.9 (0.18)	86.8 (0.17)	1.0%
PostMark	4288 (11.9)	4290 (13.0)	0.04%

Table 3: **Performance of macro benchmarks.** All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 10 trials in seconds (with the standard deviation in parenthesis).

5.1 Experimental setup

All experiments use the S4 NFS server, with and without the new support for storage-based intrusion detection. The client system is a dual 1 GHz Pentium III with 128 MB RAM, and a 3Com 3C905B 100 Mbps network adapter. The server is a dual 700 MHz Pentium III with 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro 100 Mb network adapter. The client and server are on the same 100 Mb network switch. The operating system on all machines is Red Hat Linux 6.2 with Linux kernel version 2.2.14.

SSH-build was constructed as a replacement for the Andrew file system benchmark [16, 34]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v. 1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [18]. It creates a large number of small randomly-sized files (between 512 B and 16 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 100,000 transactions on 20,000 files, and the biases for transaction types are equal.

5.2 Performance impact

The storage IDS checks a file's rules before any operation which could possibly trigger an alert. This includes READ operations, since they may change a file's last access time. Additionally, namespace-modifying operations require further checks and possible updates of the non-existent names table. To understand the performance consequences of the storage IDS design, we ran PostMark and SSH-Build tests. Since our main concern is avoiding a performance loss in the case where no rule is violated, we ran these benchmarks with no relevant rules set. As long as no rules match, the results are similar with 0 rules, 1000 rules on existing files, or 1000 rules on non-existing files. Table 3 shows that the performance impact of the storage IDS is minimal. The largest performance difference is for the configure and build phases of SSH-build, which involve large numbers of namespace operations.

Benchmark	Baseline	With IDS	Change
Create	4.32	4.35	0.7%
Remove	4.50	4.65	3.3%
Mkdir	4.36	4.38	0.5%
Rmdir	4.52	4.59	1.5%
Rename file	3.81	3.91	2.6%
Rename dir	3.91	4.04	3.3%

Table 4: **Performance of micro benchmarks.** All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 1000 trials in milliseconds.

Microbenchmarks on specific filesystem actions help explain the overheads. Table 4 shows results for the most expensive operations, which all affect the namespace. The performance differences are caused by redundancy in the implementation. The storage IDS code is kept separate from the NFS server internals, valuing modularity over performance. For example, name removal operations involve a redundant directory lookup and inode fetch (from cache) to locate the corresponding inode's `watchflags` field.

5.3 Space efficiency

The storage IDS structures are stored on disk. To avoid extra disk accesses for most rule checking, though, it is important that they fit in memory.

Three structures are used to check a set of rules. First, each inode in the system has an additional two-byte field for efficient rule lookup. There is no cost for this, because the space in the inode was previously unused. Linux's ext2fs and BSD's FFS also have sufficient unused space to store such data without increasing their inode sizes. If space were not available, the reverse lookup table can be used instead, since it provides the same information. Second, for each pathname component of a rule, the reverse lookup table requires $20 + N$ bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and $N + 2$ bytes for a pathname component of length N. Third, the non-existent names table contains one entry for every file being watched that does not currently exist. Each entry consumes 274 bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and 256 bytes for the maximum pathname supported.

To examine a concrete example of how an administrator might use this system, we downloaded the open source version of Tripwire [40]. Included with it is an example rule file for Linux, containing 391 rules. We examined a SuSE 7.1 [38] desktop machine to obtain an idea of the number of watched files that actually exist on the hard drive. Of the 391 watched files, 273 existed on our example system. Using data structure sizes from above, reverse lookup entries for the watched files consume 8 KB. Entries in the non-existent name table for the remaining 118 watched files consume 32 KB. In total, only 40 KB are needed for the storage IDS.

6 Block-based storage

Although we have not fully implemented it, we have convinced ourselves that storage-based intrusion detection can also be embedded in storage devices that export a block-based interface (e.g.,

Required information	Linux ext2fs	FreeBSD FFS
Partition type, offset, size	`struct partition`	`struct disklabel`
FS block size, inode size, inode table offsets, blocks/inodes per group, etc.	`struct ext2_super_block`, `struct ext2_group_desc`	`struct fs`
Direct/indirect block pointers, file type, file size, file attributes	`struct ext2_inode`	`struct dinode`
File names, record lengths, inode numbers	`struct ext2_dir_entry`	`struct direct`

Table 5: **Important on-disk structures.** A block-based storage device needs to understand only a few on-disk file system structures in order to identify blocks associated with each file in the rule set. Attributes of files or file systems are also identified in these structures.

array controllers or standalone disk drives). Performing intrusion detection inside block-based devices requires augmenting device firmware with an understanding of the on-disk file system structure. Specifically, the device must be able to identify which blocks contain data or metadata for a given file. Further, after any write to a "watched" block, the device must be able to determine whether the write caused the file to actually change in a way that violates a rule. With these two abilities, block-based devices can include a storage IDS with the same rules (and same basic structures) described for our NFS server prototype. This section reports on relevant experiences.

In order to understand the feasibility of this approach, we wrote the necessary utility to understand the on-disk layouts of both Linux's ext2fs [5] and FreeBSD's FFS [25]. This utility identifies all disk blocks associated with a file given its full pathname—in other words, the set of blocks that should be watched to determine whether a WRITE operation might trigger a rule violation. The on-disk structures necessary to translate between a filename and the associated metadata and data blocks are shown in Table 5. Subsets of the utility allow it to determine if a particular write actually affects a given file from the rule set. For example, writes to associated directory blocks do not always affect watched files.

Based on our experiences, we conclude that this kind of extension is feasible. In particular, the code to interpret the on-disk metadata structures is straightforward in both size (less than 200 semicolons worth of C code) and implementation complexity (less than three days each for a single graduate student). In retrospect, this straightforwardness is not surprising, despite the complexity of file system code, since the vast majority of an FS implementation relates to managing modifications (e.g., integrity maintenance) and optimizing performance (e.g., allocation and cache management).

As a business model, we expect that device manufacturers would provide firmware extensions to implement a storage IDS for a particular file system. Such patches would remain valid as long as the on-disk structures do not change. For the ext2 and FFS structures described above, analysis of historical versions of Linux and FreeBSD indicate that, once deployed, these structures usually remain compatible for at least 5 years[3].

[3]New fields are added to these structures over time to accommodate changes to the file systems' functionality. However, updates that rearrange vital fields describing the on-disk layout—for example, the noncompatible modifica-

One difference between this approach and the one described in Section 4 is that it will not be possible for block-based devices to merge IDS information into the file system structures, as is done in our prototype. In particular, the `watchflags` field stored in each inode would not be used. The reverse lookup table and non-existent name table could be kept roughly as described for our prototype. As such, the memory space analysis in Section 5.3 should hold for block-based storage as well.

Object-based storage: Somewhere between block-based storage and file-based storage lies the emerging concept of object-based storage [14, 42]. Storage-based intrusion detection will be easier for storage objects than for blocks, since files will often map directly to one or more objects. Still, the object store must understand the naming structure of the file system above it in order to watch for most of the interesting warning signs.

7 Additional Related Work

Much related work has been discussed within the flow of the paper. For emphasis, we note that there have been many intrusion detection systems focused on host OS activity and network communication; Axelsson [3] recently surveyed the state-of-the-art. Also, the most closely related tool, Tripwire [19], was used as an initial template for our prototype's storage-based intrusion detection ruleset.

Perhaps the most closely related work is the original proposal for self-securing storage [35], which argued for storage-embedded support for intrusion survival. Self-securing storage retains every version of all data and a log of all requests for a period of time called the *detection window*. For intrusions detected within this window, security administrators have a wealth of information for post-intrusion diagnosis and recovery.

Such versioning and auditing complements storage-based intrusion detection in several additional ways. First, when creating rules about storage activity for use in detection, administrators can use the latest audit log and version history to test new rules for false alarms. Second, the audit log could simplify implementation of rules looking for patterns of requests. Third, administrators can use the history to investigate alerts of suspicious behavior (i.e., to check for supporting evidence within the history). Fourth, since the history is retained, a storage IDS can delay checks until the device is idle, allowing the device to avoid performance penalties for expensive checks by accepting a potentially longer detection latency.

8 Summary

A storage IDS watches system activity from a new viewpoint, which immediately exposes some common intruder actions. Running on separate hardware, this functionality remains in place even when client OSes or user accounts are compromised. Our prototype storage IDS demonstrates both feasibility and efficiency within a file server. Analysis of the additional functionality required suggests a similar storage IDS could be embedded within block-based storage devices.

tion of `struct ext2_dir_entry` to include file type information as part of the ext2fs revision 1 deployment—are rare.

References

[1] Paul Ammann, Sushil Jajodia, and Peng Liu. Rewriting histories: recovering from undesirable committed transactions. *Distributed and Parallel Databases*, **8**(1):7–40. Kluwer Academic Publishers, January 2000.

[2] Paul Ammann, Sushil Jajodia, Catherine D. McCollum, and Barbara T. Blaustein. Surviving information warfare attacks on databases. *IEEE Symposium on Security and Privacy* (Oakland, CA, 4–7 May 1997), pages 164–174. IEEE Computer Society Press, 1997.

[3] Stefan Axelsson. *Research in intrusion-detection systems: a survey*. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[4] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, **9**(2):131–152, Spring 1996.

[5] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. http://e2fsprogs.sourceforge.net/ext2intro.html.

[6] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 273–287. USENIX Association, 2000.

[7] Peter M. Chen and Brian D. Noble. When virtual is better than real. *Hot Topics in Operating Systems* (Elmau, Germany, 20–22 May 2001), pages 133–138. IEEE Comput. Soc., 2001.

[8] B. Cheswick and S. Bellovin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley, Reading, Mass. and London, 1994.

[9] DEF CON. http://www.defcon.org/.

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

[11] Dorothy E. Denning. *Information warfare and security*. Addison-Wesley, 1999.

[12] Dan Farmer. What are MACtimes? *Dr. Dobb's Journal*, **25**(10):68–74, October 2000.

[13] Stephanie Forrest, Setven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for UNIX processes. *IEEE Symposium on Security and Privacy* (Oakland, CA, 6–8 May 1996), pages 120–128. IEEE, 1996.

[14] 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.

[15] 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.

[16] 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.

[17] Y. N. Huang, C. M. R. Kintala, L. Bernstein, and Y. M. Wang. Components for software fault-tolerance and rejuvenation. *AT&T Bell Laboratories Technical Journal*, **75**(2):29–37, March-April 1996.

[18] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[19] 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, VA, 2–4 November 1994), pages 18–29, 1994.

[20] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. *IEEE Symposium on Security and Privacy* (Oakland, CA, 04–07 May 1997), pages 175–187. IEEE, 1997.

[21] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. *USENIX Annual Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 95–106. USENIX Association, 1995.

[22] Peng Liu, Sushil Jajodia, and Catherine D. McCollum. Intrusion confinement by isolation in information systems. *IFIP Working Conference on Database Security* (Seattle, WA, 25–28 July 1999), pages 3–18, 2000.

[23] Teresa F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. *IEEE Symposium on Security and Privacy* (Oakland, CA, 18–21 April 1988), pages 59–66. IEEE, 1988.

[24] McAfee NetShield for Celerra. EMC Corporation. http://www.emc.com/pdf/partnersalliances/einfo/McAfee_netshield.pdf.

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

[26] NFR Security. http://www.nfr.net/.

[27] John Phillips. *Antivirus scanning best practices guide*. Technical report 3107. Network Appliance Inc. http://www.netapp.com/tech_library/3107.html.

[28] Phillip A. Porras and Peter G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. *National Information Systems Security Conference*, pages 353–365, 1997.

[29] Wojciech Purczynski. GNU fileutils – recursive directory removal race condition. BugTraq mailing list, 11 March 2002.

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

[31] Joel Scambray, Stuart McClure, and George Kurtz. *Hacking exposed: network security secrets & solutions*. Osborne/McGraw-Hill, 2001.

[32] Bruce Schneier. *Secrets & lies: digital security in a networked world*. John Wiley & Sons, 2000.

[33] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, **2**(2):159–176. ACM, May 1999.

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

[35] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[36] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *USENIX Annual Technical Conference* (Boston, MA, 25–30 June 2001), pages 1–14. USENIX Association, 2001.

[37] Sun Microsystems. *NFS: network file system protocol specification*, RFC–1094, March 1989.

[38] SuSE Linux 7.1. SuSE AG. ftp://ftp.suse.com/pub/suse/i386/7.1/.

[39] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5), 1995.

[40] Tripwire Open Souce 2.3.1. http://unc.dl.sourceforge.net/sourceforge/tripwire/tripwire-2.3.1-2.tar.gz.

[41] Kalyanaraman Vaidyanathan, Richard E. Harper, Steven W. Hunter, and Kishor S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Cambridge, MA, 16–20 June 2002). Published as *Performance Evaluation Review*, **29**(1):62–71. ACM Press, 2002.

[42] Ralph O. Weber. SCSI Object-Based Storage Device Commands (OSD), Working Draft. ANSI Technical Committee T10, August 2002. ftp://ftp.t10.org/t10/drafts/osd/osd-r06.pdf.