On the feasibility of intrusion detection inside workstation disks

John Linwood Griffin, Adam Pennington, John S. Bucy,
Deepa Choundappan, Nithya Muralidharan, Gregory R. Ganger

CMU–PDL–03–106
December 2003

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Storage-based intrusion detection systems (IDSes) can be valuable tools in monitoring for and notifying administrators of malicious software executing on a host computer, including many common intrusion toolkits. This paper makes a case for implementing IDS functionality in the firmware of workstations' locally attached disks, on which the bulk of important system files typically reside. To evaluate the feasibility of this approach, we built a prototype disk-based IDS into a SCSI disk emulator. Experimental results from this prototype indicate that it would indeed be feasible, in terms of CPU and memory costs, to include IDS functionality in low-cost desktop disk drives.

Keywords: Computer security, intrusion detection, IDS, local disk, disk firmware.

1 Introduction

Intrusion detection systems (IDSes) are important tools for identifying suspicious and malicious activity in computer systems. Such IDSes are commonly deployed both on end-user computers (hosts) and at points in the network leading to hosts. As with most things, however, there are no perfect intrusion detection systems. All are susceptible to false positives and undetected intrusions. Most also are much better at detecting probes and attempts to intrude than they are at detecting misbehavior after an intrusion. For example, host-based IDSes are vulnerable to being turned off when the host is compromised, and many fewer signs are visible to network-based IDSes once the intruder is "in."

A storage-based IDS runs inside of a storage device, watching the sequence of requests for signs of intrusions [Pennington03]. Storage-based IDSes are a new vantage point for intrusion detection, offering complementary views into system activity, particularly after a successful intrusion has begun. Many rootkits and intruders manipulate files that can be observed by the disk that stores them. For example, an intruder may overwrite system binaries or alter logfiles in an attempt to hide evidence of the intrusion. Other examples include adding backdoors, Trojan horses, or discreet repositories of intruder content (such as pirated content). A storage-based IDS can detect such actions. Further, because a storage-based IDS is not controlled by the host operating system (or by the host's IDS), it will continue to operate even when the network-based IDS is circumvented and the host-based IDS is turned off.

Our previous work developed and experimented with a storage-based IDS in the context of a NFS server [Pennington03], illustrating that it involves minimal overhead and can detect many intrusions in diskless hosts (i.e., hosts with all system files on the NFS server). However, most environments don't work that way. The vast majority of systems—and those that are most vulnerable to attack—are single-user systems with local disks. To be effective in practice, it must be

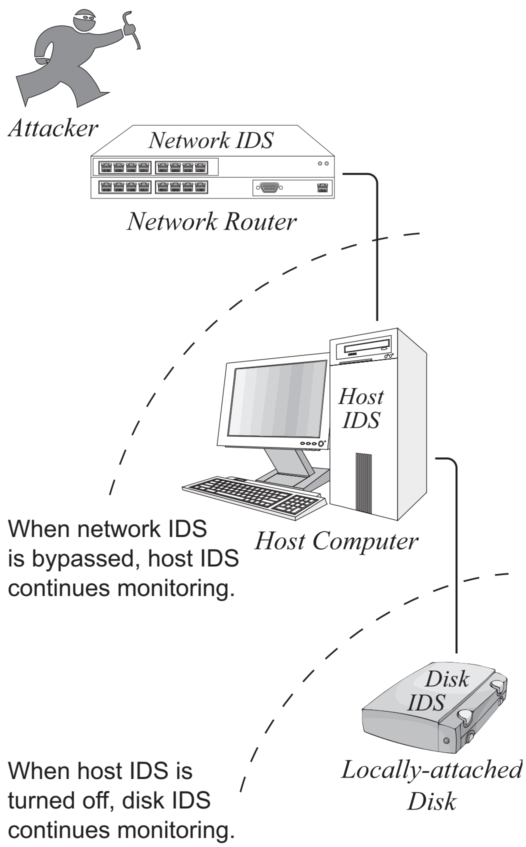


Figure 1: **The role of a disk-based intrusion detection system (IDS).** *A disk-based IDS watches over all data and executable files that are persistently written to local storage, monitoring for suspicious activity that might indicate an intrusion on the host computer.*

possible to run the storage-based IDS on the local disk of each workstation. Figure 1 shows an example of such a disk-based IDS deployment. The host computer is a standard user desktop and is therefore vulnerable to user errors and software holes. The storage-based IDS in the disk backs up the host and network IDSes and runs on a SCSI or IDE/ATA disk with expanded processing capabilities. All three IDSes are administered by an external administrative machine, perhaps as part of a larger-scale managed computing environment.

A workstation disk is a challenging environment for intrusion detection. To function, a disk-based IDS must have semantic knowledge about the file systems contained on the disk, so it can analyze low-level "read block" and "write block" requests for actions indicating suspicious accesses or modifications to the file system. At the same time, disks are embedded devices with cost-constrained processing and memory resources and little tolerance for performance degradation. Additionally, effective operation will require a secure channel for communication between the administrator and the disk-based IDS; for practical purposes, this channel should seamlessly integrate with the disk's existing block interface.

This paper makes a case for the feasibility of performing intrusion detection inside workstation disks. It describes a prototype disk-based IDS built on top of a storage device emulator. It connects to the SCSI bus in a real computer system, looking and "feeling" just like a disk, and monitors real storage requests for behavior that matches the behavior of real-world intrusion tools. Experiments with this prototype demonstrate that the CPU and memory costs will be well within tolerable bounds for modern desktop disks for reasonable rulesets and workloads. For example, only a few thousand CPU cycles are needed for over 99% of disk I/Os, and less than a megabyte is needed even for aggressive rulesets. Moreover, any disk in which the IDS is not enabled would incur no CPU or memory cost, making it feasible to include the support in all devices and enable only for those that pay for licenses; this model has worked well for 3com's NIC-based firewall product [3Com01a].

The remainder of this paper is organized as follows. Section 2 motivates workstation disks as an untapped location for real-time intrusion detection. Section 3 discusses IDS design challenges relating to the workstation disk environment. Section 4 describes the prototype disk-based IDS system. Section 5 evaluates the prototype performance. Section 6 discusses the feasibility of real-world IDS integration. Section 7 summarizes the contributions of this work.

2 Background and motivation

This section motivates the use of workstation disks as a target environment for intrusion detection, discusses previous work that demonstrates the effectiveness of a storage-based IDS in identifying real-world intruder behavior, and reviews work related to storage-based IDSes.

2.1 Intrusion detection in storage

Storage-based intrusion detection enables storage devices such as workstation disks to watch storage traffic for suspicious behavior. Because storage-based IDSes run on separate hardware with a limited external interface, they enjoy a natural compromise independence from the hosts to which they are attached: An attacker who breaches the security of the host must then breach a separate perimeter to disable a security system on a storage device. Also, because storage devices see all persistent activity on a computer, several common intruder actions [Denning99, p. 218][Scambray01, pp. 363–365] are quite visible inside the storage device.

A storage IDS shines when other IDSes have been bypassed or disabled but neither the storage device nor the administrator's computer have been compromised. The threat model we address is when an attacker has full software control but not full hardware control over the host system. This could come in the form of an intruder breaking into a host over the network, or from a user with administrative privileges mistakenly executing malicious software. We do not explicitly protect against an aggressive insider with physical access to the disk, although Section 3.2 discusses a possible solution to this problem.

Administrators can watch for a variety of possible intruder actions inside a storage-based IDS [Pennington03]. First, the IDS could watch for unexpected modifications to file data or metadata; this is similar to the functionality of Tripwire [Kim94a], except that a storage-based IDS monitors in real time. The IDS could also watch for suspicious access patterns, such as a non-append write to a system logfile. It could watch for loss of file integrity for well-structured files such as `/etc/passwd` or database files. And it could watch for suspicious content, such as malformed file names or known virus signatures.

Storage-based intrusion detection is not a panacea. While false positives should be infrequent, legitimate actions that modify a watched file will create an alert to the administrator. The broader the scope of watched actions, the higher the frequency of false positives will be. On the other hand, a storage IDS could miss an intrusion entirely if it is configured to watch too limited a set of files. Additionally, a storage-based IDS will likely have some performance impact on the host computer's workload.

2.2 Real-world efficacy of a storage IDS

In previous work, Pennington et al. [Pennington03] built and analyzed a storage-based IDS inside an NFS server. Their work demonstrates that storage-based intrusion detection is indeed an effective tool for detecting the effects of real-world attacks, and that their implementation is efficient in its processing and memory requirements.

To evaluate the real-world efficacy of their server-based IDS, the authors analyzed the behavior of eighteen publicly-available intrusion tools. They found that 83% of the intrusion tools modified one or more system files, and that these modifications would be noticed by their storage-based IDS when it monitored a simple, default ruleset. Also, 39% of the tools altered a system log file, an action which also would be detected by the storage-based IDS under the default ruleset. When the authors analyzed a real host computer that had been unexpectedly compromised, they discovered that a storage-based IDS would have immediately noticed the intrusion because of the system binaries that were modified during the attack.

The authors also analyzed some brief traces of a desktop workstation's disk and reported preliminary results indicating that false positives would very rarely be reported by a storage-based IDS, with the exception of (planned) nightly rewrites of the password file. Section 5.4 presents results from a more extensive collection of traces, confirming and underscoring this result.

2.3 Related work

Intrusion detection is a well-studied field. One of the earliest formalizations of intrusion detection is presented by Denning et al. [Denning87a]. Tripwire is one of the more well-known intrusion detection systems [Kim94a, Kim94b]; we use the suggested Tripwire configuration as part of the basis for the ruleset in the experiments in this paper. Investigations have been made into intrusion detection systems founded on machine learning [Forrest96] as well as static rules to watch system calls on a machine [Ko97a]. In 1998, Axelsson surveyed the state-of-the-art for intrusion detection systems [Axelsson98].

Recent research has explored other ways of creating similar protection boundaries to ours. Chen and Noble [Chen01a] and Garfinkel and Rosenblum [Garfinkel03] propose using a virtual machine monitor (VMM) that can inspect and observe machine state while remaining compromise independent of most host software. This could be expanded by adding a storage-based IDS into the virtual machine's storage module. Additionally, recent work explores the idea of hardware support for doing intrusion detection inside systems [Ganger01, Zhang02a].

Adding IDS functionality to storage devices can be viewed as part of a recurring theme of migration of processing capability to peripheral devices. For example, several research groups have explored the performance benefits of offloading scan and other database-like primitives to storage devices [Acharya98, Keeton98, Riedel98]. Other research has explored the use of device intelligence for eager writing [Chao92, Wang99]. Recently, Sivathanu et al. proposed the general notion of having storage devices understand host-level data semantics and use that knowledge to build a variety of performance, availability, and security features [Sivathanu03]. A disk-based IDS is a specific instance of such "semantically-smart disk systems."

3 Design issues for disk-based intrusion detection systems

There are four main design issues for a storage-based intrusion detection system [Pennington03]. These include specifying access policies, securely administering the IDS, monitoring storage activity for policy violations, and responding to policy violations. This section discusses the aspects of these that specifically relate to the challenging environment of workstation disks.

3.1 Specifying access policies

For the sake of of usability and correctness, there must be a simple and straightforward syntax for human administrators to state access policies to a disk-based IDS. Although a workstation disk operates using a block-based interface, it is imperative that the administrator be able to refer to higher-level file system objects contained on the disk when stating policies. As an example, an appropriate statement might be: *Warn me if anything changes in the directory /sbin.* In our experience, the Tripwire-like rules used by Pennington et al. to specify access policies for their server-based IDS [Pennington03] work well for specifying policies to a disk-based IDS.

A disk-based IDS must be capable of mapping such high-level statements into a set of violating interface actions. This set of violating actions may include writes, reads (e.g., of honeytokens [Card] such as `creditcards.txt`), and interface-specific commands (such as the FORMAT UNIT command for SCSI). One such mapping for the above "no-change" rule for `/sbin` could be: *Generate the alert "the file /sbin/fsck was modified" when a write to block #280 causes the* *contents of block #280 to change.* To accomplish this mapping, the IDS must be able to read and interpret the on-disk structures used by the file system. However, the passive nature of intrusion detection means it is not necessary for a disk-based IDS to be able to modify the file system, which simplifies implementation.

Workstation disks are frequently powered down. These mappings must be restored to the IDS (for example, from an on-disk copy) before regular operation commences. We anticipate this requiring perhaps a few megabytes of private disk space. This approach is no different from the other tables kept by disk firmware, such as for tracking defective sectors and predicting service times efficiently.

A disk-based IDS is capable of watching for writes to free space that do not correspond with updates to the file system's block allocation table. Storing object and data files in unallocated disk blocks is one method used by attackers to hide evidence of a successful intrusion [Grugg02]. Such hidden files are difficult to detect, but are accessible by processes (such as an IRC or FTP server) initiated by the attacker. Depending on the file system, watching free space may cause extra alerts to be generated for short-lived files which are deleted after the contents are written to disk but before the allocation table is updated.

3.2 Disk-based IDS administration

For effective real-time operation of a disk-based IDS, there must be a secure method for communication between an administrator's computer and the IDS. This communication includes transmitting access policies to the IDS, receiving alerts generated by the IDS, and acknowledging the receipt of policies and alerts. Unlike in a server-based environment, however, a workstation disk will likely not have direct access to a communications network to handle such administrative traffic. In this case, this traffic must be routed through the host and over the existing physical link

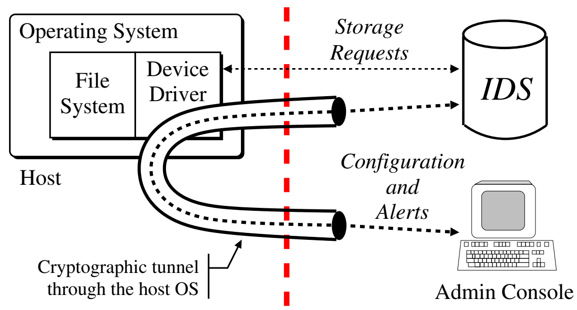


Figure 2: **Tunneling administrative commands through client systems.** *For disks attached directly to host computers, a cryptographic tunnel allows the administrator to securely manage a disk-based IDS. This tunnel uses untrusted software on the host computer to transport administrative commands and alerts.*

connecting the host with the disk, as shown in Figure 2. In other words, the host must be actively involved in bridging the gap between the administrator and the disk.

This communications model presents several challenges. For one, in most block-based storage interconnects, the disk takes the role of a passive target device and is unable to initiate a data transfer to the administrator's computer. Instead, the administrator must periodically poll the IDS to query if any alerts have been generated. Also, the administrative communication path is vulnerable to disabling by an attacker who has compromised the host system. In response to such disabling, both the administrator and the IDS could treat a persistent loss of communication as a policy violation. The IDS could alter its behavior as described in Section 3.4, while additionally logging any subsequent alerts for later transmission to the administrator.

The communications channel between the disk-based IDS and the administrator must be protected both from eavesdropping and tampering by an attacker. Such a secure channel can be implemented using standard cryptographic techniques. For the network intruder and rogue software concerns in our security model, it is sufficient for secret keys to be kept in the disk's firmware. If physical attacks are an issue, disk secure coprocessors can be used [Zhang02a]. Such additions are not unreasonable for future system components [Gobioff99, Lie00].

3.3 Monitoring for policy violations

Once the administrative policy is received by a disk-based IDS, all storage requests arriving at the disk should be checked against the set of violating interface actions. A check should be performed for every block in a request: a write to block 72 of length 8 blocks should check for violating actions on any of blocks 72–79. As this check is in the critical path of every request, it should be implemented as efficiently as possible.

Some file system objects, such as the directory listing for a directory containing many files, will span multiple sequential disk blocks. Space for such objects will generally be allocated by the file system in a multiple of the file system block (hereafter, *fs-block*) size. We found it convenient for our disk-based IDS prototype to evaluate incoming requests by their impact on entire fs-blocks instead of on individual disk blocks: *Generate the alert "the file /sbin/fsck was modified" when a write to fs-block #35 causes the contents of fs-block #35 to change.* When monitoring at the level of fs-blocks, a disk-based IDS may need to momentarily queue several pending requests (all of which affect a single fs-block) in order to evaluate their combined effect on the fs-block atomically.

When checking the legality of write requests, a disk-based IDS may need to first fetch the previously written (old) data for that block from the disk. The old data can then be compared with the new data in the write request to determine which bytes, if any, are actually changed by the write. Such a check would also be necessary when a block contains more than one file system object—for example, a single block could contain several file fragments or inode structures—and different administrative policies apply to the different objects. A similar check allows the system to quell alerts that might otherwise be generated during file system defragmentation or reorganization operations; no alerts are generated unless the actual contents or attributes of the files are modified.

3.4 Responding to policy violations

For an intrusion detection system in a workstation disk, the default response to a policy violation should be to prepare an administrative alert while allowing the request to complete. This is because the operating system may halt its forward progress when a locally-attached disk returns an error or fails to complete a request, especially at boot time or during crash recovery.

Pennington et al. discuss several other possible responses for a storage-based IDS after a policy violation [Pennington03]. These include artificially slowing storage requests while waiting for an administrative response, and internally versioning all subsequent modifications to aid the administrator in post-intrusion analysis and recovery [Strunk00]. To support versioning without needing to modify the file system, a disk-based IDS can copy the previous contents of written blocks to a private area on the disk that is inaccessible from the host computer. The administrator can later read the contents of this area over the administrative communication channel.

After a policy violation, the IDS should make note of any dynamic changes to the on-disk file system structure and adjust its behavior accordingly. For example, for the policy specified in Section 3.1, when a new file is created in the `/sbin` directory the system should first generate an alert about the file creation and then begin monitoring the new file for changes.

4 Prototype implementation

We built a prototype disk-based IDS called *IDD* (Intrusion Detection for Disks). The IDD takes the form of a PC masquerading as a SCSI disk, enhanced with storage-based intrusion detection. From the perspective of the host computer, the IDD looks and behaves like an actual disk. This section describes architectural and implementation details of this prototype.

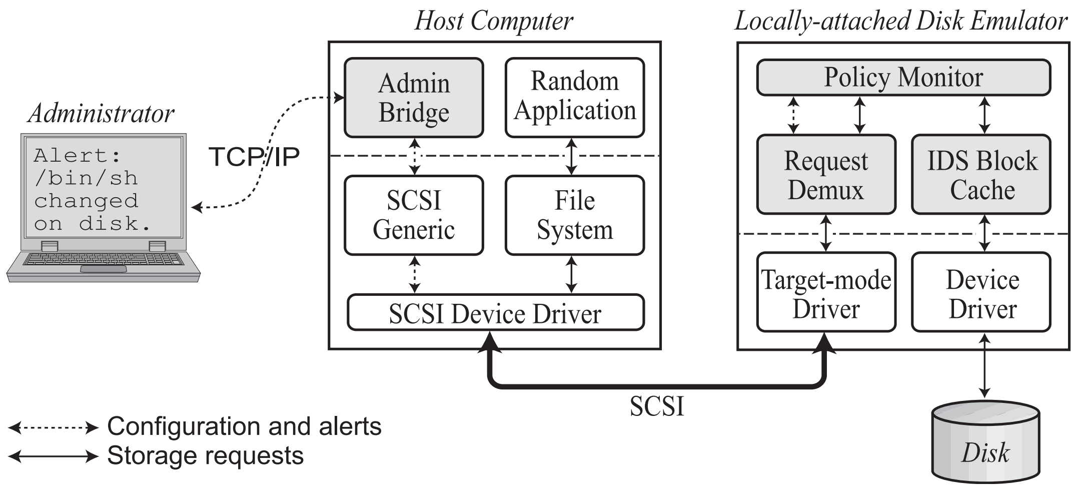


Figure 3: **Disk-based IDS prototype architecture.** *This figure shows the communications flow between a Linux-based host computer and the locally-attached, FreeBSD-based IDD. The shaded boxes are key components that support disk-based IDS operation. Ordinary storage traffic is initiated by application processes, passes across the SCSI bus, is checked by the policy monitor, and is finally serviced by the disk. Administrative traffic is initiated by the administrator, passes across a TCP/IP network, is received by the bridge process on the host computer, passes across the SCSI bus, and is finally serviced by the policy monitor. The sample alert displayed on the administrator's console originated in the policy monitor.*

4.1 Architecture

Figure 3 shows the high-level interactions between the IDD, the administrator, and the intermediary host computer. The three major components that implement the disk-based IDS functionality are the bridge process on the host computer and the request demultiplexer and policy manager on the IDD.

The bridge process forwards commands from the administrator to the IDD and conveys administrative alerts from the IDD to the administrator. The request demultiplexer identifies which incoming SCSI requests contain administrative data and handles the receipt of administrative policy and the transmission of alerts. Together, these components implement the administrative communications channel, as described in Section 4.2.

The policy monitor handles the mapping of administrative policy into violating interface actions. It also monitors all ordinary storage traffic for real-time violations, and generates alerts to be fetched by the administrator. This management of administrative policies is discussed further in Section 4.3.

4.2 Administrative communication

The administrative communications channel is implemented jointly by the bridge process and the request demultiplexer. The administrator sends its traffic directly to the bridge process over a TCP/IP-based network connection. The bridge process immediately repackages that traffic in the form of specially-marked SCSI requests and sends those across the SCSI bus. When these marked requests arrive inside the IDD, they are identified and intercepted by the request demultiplexer. Encryption of messages is handled by the administrator's computer and the request demultiplexer.

The repackaging in the bridge process takes different forms depending on whether the administrator is sending new policies (outgoing traffic) or polling for new alerts (incoming traffic). For outgoing traffic, the bridge creates a single SCSI WRITE 10 request containing the entire message. The request is marked as containing administrative data by setting a flag in the SCSI command descriptor block[1]. The request is then sent to the bus using the Linux SCSI Generic passthrough driver interface.

For incoming traffic, the bridge creates either one or two SCSI READ 10 requests. The first request is always of fixed-size (we used 8 KB) and is used to determine the number of bytes of alert data waiting in the IDD to be fetched: The first 32 bits received from the IDD indicate the

[1] We set bit 7 of byte 1 in the WRITE 10 and READ 10 command descriptor blocks. This bit is otherwise unused and is marked as reserved in the SCSI-3 specification. This method was the simplest of several options we considered—other options included using vendor-specific operation codes for administrative reads and writes, or using MODE SENSE and MODE SELECT to access administrative mode pages.

integer number of pending bytes. The remaining space in the first request is filled with waiting data. If there is more data waiting than fits in the first request, a second request immediately follows. This second request is of appropriate size to fetch all the remaining data. These requests are marked as containing administrative data and sent in the same manner as for outgoing traffic. Once the bridge has fetched all the waiting data, it forwards the data to the administrator over the network.

Outgoing and incoming messages contain sequence and acknowledgment numbers, to ensure that policies or alerts are not mistakenly or otherwise dropped. Our implementation sends one pair of messages (outgoing and incoming) per second by default. In order to reduce administrator-perceived lag, this frequency is temporarily increased whenever recent messages contained policies or alerts.

4.3 Administrative policy management

The policy monitor bridges the semantic gap between the administrator's policy statements and violating interface actions, and it audits all storage traffic from the host computer in real time. The policy monitor operates at the file system block (fs-block) level, as discussed in Section 3.3. Request block numbers are converted to the relevant partition and fs-block numbers upon request arrival. (Hereafter in this section, "block" refers to a fs-block.)

The IDD has a relatively simple semantically-smart [Sivathanu03] understanding of the on-disk file system structures. The IDD currently understands the ext2 file system used by Linux-based host computers [Card94a]; to support this we hard-coded the structure of on-disk metadata into the policy manager. For ext2 this included the ext2 superblock, inode, indirect block, and directory entry structures. As an alternative to hard-coding, we envision a more flexible administrative interface over which the relevant file system details could be downloaded.

As administrative policy is specified, the IDD receives a list of files whose contents should be watched in particular ways (e.g., for any change, for reads, and for non-append updates). For each of these watched files, the IDD traverses the on-disk directory structure to determine which metadata and data blocks are associated with the file. Each such block is then associated with an access check function (ACF) that can evaluate whether a block access violates a given rule. For example, the ACF for a data block and the "any change" policy would simply compare the old contents of the block with the new. The ACF for an inode block and the "non-append updates" policy would compare the old and new contents to ensure that the access time field and the file size field only increased. The ACF for a directory entry block and the "any change" policy would check the old and new contents to determine if a particular filename-to-inode-number mapping changed (e.g., `mv file1 file2` when `file2` is watched) and, if so, that the new inode's file contents match the old inode's file contents.

After a block is associated with an ACF, it is added to the watched block table (WBT). The WBT is the primary structure used by the policy manager for storage request auditing. It is stored in private, reserved disk space for persistence and paging. A WBT entry contains a monitored block number, a pointer to the associated ACF, and a human-understandable explanation (e.g., *the contents of the file /bin/netstat were modified*) to be sent whenever the ACF reports a violation. The IDD maintains a separate WBT for each monitored partition, as well as a WBT for the partition table and other unpartitioned disk space.

When a storage request arrives from the host computer, its blocks are checked against the appropriate partition's WBT. If any blocks are indeed watched, the associated ACFs for those blocks are invoked to determine whether an alert should be generated. As discussed in Section 3.3, checking the validity of a write request may require the old data to be read from the disk. Blocks that have a high probability of being checked, such as those containing monitored inodes and directory entries, are cached internally by the IDD to reduce IDS-related delays. Note that this cache is primarily needed to quickly execute ACFs on block updates that will not generate an alert; generally, we are unconcerned with performance when alerts are to be generated. Examples of appropriate cache uses include directory blocks in which a subset of names are watched and inode blocks in which a subset of inodes are watched.

5 Evaluation

This section examines the performance and memory overheads incurred by IDD, our prototype disk-based IDS. As hoped, we find that these overheads are not unreasonable for inclusion in workstation disks.

5.1 Experimental setup

Both the host computer and the locally-attached disk emulator are 2 GHz Pentium 4-based computers with 512 MB RAM. The disk emulator runs the FreeBSD 5.1 distribution and makes use of the FreeBSD target-mode SCSI support in order to capture SCSI requests initiated by the host computer. The host computer runs the Red Hat Linux 7.3 distribution. The machines are connected point-to-point using QLogic QLA2100 fibre channel adapters. The backing store disk on the emulator is a Fujitsu MAN3184MP connected to an Adaptec 29160N Ultra160 SCSI controller. In an effort to exercise the worst-case storage performance, the disk emulator was mounted synchronously by the host computer and caching was turned off inside the backing store disk.

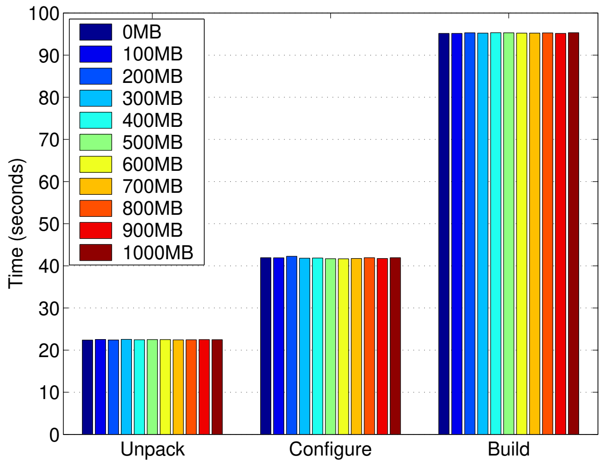
We do not argue that embedded disk processors will have a 2 GHz clock frequency; this is perhaps an order of magnitude larger than one might expect. However, an actual disk-based IDS would be manually tuned to the characteristics of the disk it runs on and would therefore run more efficiently than the IDD, perhaps by as much as an order of magnitude. To compensate for this uncertainty, we report processing overheads both in elapsed time and in processor cycle counts, the latter of which provides a reasonably portable estimate of the amount of work performed by the IDD.

Our experiments use microbenchmarks and two macrobenchmarks: PostMark and SSH-build. The PostMark benchmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [Katcher97]. 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 10,000 transactions on 200 files, and the biases for transaction types are equal.

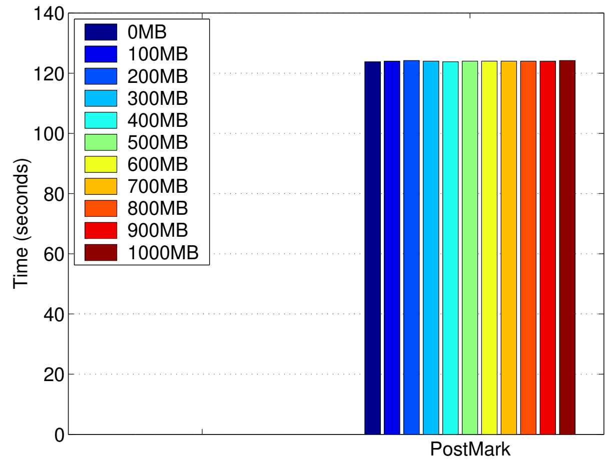
The SSH-build benchmark[Seltzer00] was constructed as a replacement for the Andrew file system benchmark [Howard88]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v. 1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on 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.

5.2 Common-case performance

When new requests arrive from the host computer, the IDD checks whether any of the request's blocks are in the WBT. In the (very) common case, no matches are found. This means there are no policy implications for that request, so no further IDS processing is required.



(a) SSH-build benchmark



(b) Postmark benchmark

Figure 4: **Application benchmarks.** *These graphs show the impact of the initial WBT check on application performance. These experiments were run with the IDS engaged and watching different amounts of data; the file system was constructed such that no policy violations were generated by the disk accesses. The data indicate virtually no scaling of the overhead as a function of the amount of watched data. 0 MB is the leftmost thin bar in each group.*

As shown in the second column of Table 1, the WBT lookup takes very little time and requires less than 1500 cycles. As expected, the added latency has a minimal effect on application performance: Figures 4(a) and 4(b) show that the application run times for PostMark and SSH-build do not change when varying the number of monitored blocks over a large range.

The overheads of the IDS infrastructure itself are also small. The difference in PostMark run times between running the disk emulator with no IDS and running it with an empty IDS (with no policy set, and therefore no WBT lookup) was less than 1%.

The size of the WBT is approximately 20 bytes per watched block. To put this in context, Tripwire's default ruleset for Red Hat Linux expanded on our testbed host to 29,308 files, which in turn translates to approximately 225,000 watched file system blocks. This would require 4.5 MB to keep the entire WBT in core. This number can be substantially reduced, with a small slowdown for

Old data status	WBT check time	Fetch old data	ACF check time	Disk access time	Total time
Data block write: Changes the data in one block of a monitored file.					
Not cached	0.655 μs (0.076 μs) 1310 cycles	**4,210** μs (1,580 μs)	0.447 μs (1.48 μs) 893 cycles	**5,840** μs (21.0 μs)	10.1 ms
Cached	0.554 μs (0.102 μs) 1110 cycles	3.21 μs (0.340 μs) 6410 cycles	0.270 μs (0.033 μs) 539 cycles	**4,390** μs (1,500 μs)	4.39 ms
Metadata block write: Changes the last-modify-time in a monitored inode.					
Not cached	0.548 μs (0.094 μs) 1100 cycles	**3,860** μs (1,420 μs)	0.250 μs (0.029 μs) 501 cycles	**5,700** μs (716 μs)	9.56 ms
Cached	0.594 μs (0.134 μs) 1190 cycles	3.07 μs (0.477 μs) 6140 cycles	0.772 μs (2.59 μs) 1540 cycles	**3,810** μs (1,780 μs)	3.81 ms

Table 1: **Microbenchmarks.** *This table decomposes the service time of write requests that set off rules either on file data or metadata. The numbers in parentheses show the standard deviation of the average. Two cases for each are shown, where the requested blocks either are or are not already present in the IDD block cache. With caching enabled, the total time is dominated by the main disk access. When blocks are not cached, the service time is roughly doubled because an additional disk access is required to fetch the old data for the block. The three phases of an IDS watched block evaluation are described in Section 4.3. Numbers shown in bold represent the dominating times in these experiments.*

the non-common-case IDS performance, by only keep 4 bytes per watched block in core (or, better yet, keeping lists extents of watched blocks in core) and demand paging the remainder of the WBT. This would reduce the memory cost to 900 KB or less. (At time of this writing, our prototype does not yet demand page the WBT.)

5.3 Additional checks on watched blocks

When one or more of a write request's blocks are found in the WBT, additional checks are made to determine whether the request violates administrative policy. If necessary, the old data is fetched from the disk in order to determine which bytes are changed by the write. The ACF is then executed; if this determines that the request illegally modifies the data, then an alert is generated and queued to be sent to the administrator.

We used microbenchmarks to measure the performance of various types of rules being matched, which we show in Table 1. Each alert was triggered 200 times, accessing files at random which had rules set for them. In order to determine if a rule has been violated, and to send an alert, several operations need to take place. For any write to a file data block, for example, the IDD needs to determine first if the given block is being watched; this takes 0.665 μs. It then reads the modified block on the disk to see if the write results in a modification; this takes 4,207 μs if the IDD does not have the block to be modified in cache, and 3.208 μs if it does. If the write is to an inode block, it takes 0.25 μs to determine if it changes a watched field in that inode. Finally, an alert is generated and the write is allowed to complete, this takes 5,841 μs if the block wasn't in cache (one revolution from the completion of the read request of the same block), and 4,386 μs if it was in cache. Table 1 summarizes the time taken by all of these actions, as well as the number of CPU cycles. We view these numbers as conservative estimates of CPU costs, because the IDS code is untuned and known to have inefficiencies. Nonetheless, the results are quite promising.

5.4 Frequency of IDS invocation

To understand the frequency of overheads beyond the common-case performance, we examined 11 months worth of local file system traces from managed desktop machines in CMU's ECE Department. The traces included 820,145,133 file operations, of which 1.8% were modifications to the disk. Using these traces, we quantify the frequency of two cases: actual rule violations and non-rule-violating updates (specifically, incidental updates to shared inode blocks or directory blocks).

We examine the traces for violations of the ruleset used by Pennington et al. [Pennington03], which includes Tripwire's default rule set for Red Hat Linux and a few additional rules regarding hidden names and append-only audit logs. This expanded out to rules watching 29,308 files, which in turn translates to approximately 225,000 blocks being watched. In the traces, 5350 operations (0.0007% of the total) impacted files with rules set on them. All but 10 of these were the result of nightly updates to configuration files such as `/etc/passwd` and regular updates to system binaries. As discussed in [Pennington03], a method for coordinating administrative updates with the IDD would convert the response to planned updates from alerts to confirmations.

The first class of non-rule-violating updates that require ACF execution is shared inode blocks. Our prototype notices any changes to an inode block containing a watched inode, so it must also determine if any given modification impacts an inode being watched. In the case of the ext2 file system[Card94a], 32 inodes share a given block. If any inode in a given block is watched, an update to one of the 31 remaining inodes will incur some additional overhead. To quantify this effect, we looked at the number of times an inode was changed which was in the same block as a watched inode. For this analysis, the local file systems of 15 computers were used as examples of which inodes share blocks with watched files. For our traces, 1.9% of I/Os resulted in changes to inode blocks. Of these, 8.1% update inode blocks that are being watched (with a standard deviation of 2.9% over the 15 machines), for a total of 0.15% of I/Os requiring ACF execution. Most of the inode block overlap resulted from these machines' `/etc/passwd` being updated nightly. This caused its inode to be in close proximity with many short-lived files in `/tmp`. On one machine, which had its own partition for `/tmp`, we found that only 0.013% of modifications caused writes to watched inode blocks. Using the values from Table 1, we compute that the extra work would result in a 0.01–0.04% overhead (depending on the IDD cache hit rate).

Similarly, the IDD needs to watch directories between a watched file and the root directory. We looked at the number of namespace changes the IDD would have to process given our traces. Using the same traces, we found that 0.22% of modifications to the file system result in namespace changes that an ACF would need to process in order to verify that no rule was violated. Based on the Table 1 measurements, these ACF invocations would result in a 0.004–0.01% performance impact, depending on IDD cache hit rate.

6 Discussion of feasibility

Workstation disks are extremely cost-sensitive components, making feature extensions a tough proposition. Security features, however, are sufficiently important and marketable today that feature extensions are not impossible. To make for a viable business case, uninterested customers must observe zero cost. the cost of any hardware support needed must be low enough that The profits from the subset of customers utilizing (and paying for) the IDS features must compensate for the marginal hardware costs incurred on all disks produced. A similar situation exists in the network interface card (NIC) industry, where 3com embedded sufficient hardware in their standard NICs to allow them to sell firewall-on-NIC extensions to the subset of interested security-sensitive customers [3Com01a]; the purchased software is essentially an administrative application that enables the support already embedded in each NIC [3Com01b] plus per-seat licenses.

Evaluation of our disk-based IDS prototype suggests that IDS processing and memory requirements are not unreasonable. In the common case of no ACF invocations, even with our untuned code, we observe just a few thousand cycles per disk I/O. Similarly, for a thorough ruleset, the memory required for IDS structures and sufficient cache to avoid disk reads for non-alert ACF executions (e.g., shared inode blocks) is less than a megabyte. Both are within reasonable bounds for modern disks. They may slightly reduce performance, for example by reducing the amount of disk cache from 2–8 MB (representative values for today's ATA drives) by less than one megabyte. The overall effect of such changes should be minor in practice, since host caches capture reuse while disk caches help mainly with prefetching. Moreover, neither the memory nor the CPU costs need be incurred by any disk that does not actually initialize and use its IDS functionality.

In addition to the IDS functionality, a disk-based IDS requires the disk to be able to perform the cryptographic functions involved with the secure administrative channel. This requires a key management mechanism and computation support for the cryptography. Again referring to the 3com NIC example, these costs can be very small. Further, various researchers have proposed the addition of such functionality to disks to enable secure administration of access control functions [Aguilera03, Gobioff99], and it can also be used to assist secure bootstrapping [Arbaugh97].

7 Summary

Storage-based intrusion detection is a promising approach, but it would be most effective if embedded in the local storage components of individual workstations. From experiences developing and analyzing a complete disk-based IDS, implemented in a disk emulator, we conclude that such embedding is feasible. The CPU and memory costs are quite small, particularly when marginal hardware costs are considered, and would be near-zero for any disk not using the IDS functionality. The promise of enhanced intrusion detection capabilities in managed computing environments, combined with the low cost of including it, makes disk-based intrusion detection a functionality that should be pursued by disk vendors.

Acknowledgements

The authors thank Joan Digney for assisting with the figures in this paper. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, 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.[2] Mr. Griffin is supported by an Intel PhD Fellowship.

References

[3Com01a] 3Com. *3Com Embedded Firewall Architecture for E-Business*. Technical Brief 100969-001. 3Com Corporation, April 2001.

[3Com01b] 3Com. *Administration Guide, Embedded Firewall Software*. Documentation. 3Com Corporation, August 2001.

[Acharya98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998), pages 81–91. ACM, 1998.

[2]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.

[Aguilera03] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Mike Burrows Timothy Mann, and Chandramohan A. Thekkath. Block-level security for network-attached disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 159–174. USENIX Association, 2003.

[Arbaugh97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. *IEEE Symposium on Security and Privacy* (Oakland, CA, 4–7 May 1997), pages 65–71. IEEE Computer Society Press, 1997.

[Axelsson98] Stefan Axelsson. *Research in intrusion-detection systems: a survey*. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[Card] Lance Spitzner. Honeytokens: The Other Honeypot. Security Focus, 21 July, 2003. `http://www.securityfocus.com/infocus/1713`.

[Card94a] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the Second Extended Filesystem. *First Dutch International Symposium on Linux* (Amsterdam, The Netherlands, December 1994), 1994.

[Chao92] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. *Mime: high performance parallel storage device with strong recovery guarantees*. Technical report HPL-92-9. 18 March 1992.

[Chen01a] 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.

[Denning87a] D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman, and W. Shockley. A multi-level relational data model. *IEEE Symposium on Security and Privacy* (Oakland, CA, 27–29 April 1987), pages 220–234. IEEE Computer Society Press, 1987.

[Denning99] Dorothy E. Denning. *Information warfare and security*. Addison-Wesley, 1999.

[Forrest96] 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.

[Ganger01] Gregory R. Ganger and David F. Nagle. Better security via smarter devices. *Hot Topics in Operating Systems* (Elmau, Germany, 20–22 May 2001), pages 100–105. IEEE, 2001.

[Garfinkel03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. *NDSS* (San Diego, CA, 06–07 February 2003). The Internet Society, 2003.

[Gobioff99] 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.

[Grugg02] The Grugg. Defeating forensic analysis on Unix. *Phrack Magazine*, **11**(59):6–6, 28 July 2002.

[Howard88] 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.

[Katcher97] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[Keeton98] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, **27**(3):42–52, September 1998.

[Kim94a] 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. ACM, 1994.

[Kim94b] Gene H. Kim and Eugene H. Spafford. *Experiences with Tripwire: using integrity checkers for intrusion detection*. CSD-TR-94-012. 21 February 1994.

[Ko97a] 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.

[Lie00] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), pages 169–177. ACM, 2000.

[Pennington03] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: watching storage activity for suspicious behavior. *USENIX Security Symposium* (Washington, DC, 06–08 August 2003), 2003.

[Riedel98] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. *International Conference on Very Large Databases* (New York, NY, 24–27 August, 1998). Published as *Proceedings VLDB*, pages 62–73. Morgan Kaufmann Publishers Inc., 1998.

[Scambray01] Joel Scambray, Stuart McClure, and George Kurtz. *Hacking exposed: network security secrets & solutions*. Osborne/McGraw-Hill, 2001.

[Seltzer00] 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), pages 71–84, 2000.

[Sivathanu03] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically smart disk systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31–02 April 2003), pages 73–88. USENIX Association, 2003.

[Strunk00] 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.

[Wang99] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, 1999.

[Zhang02a] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. *ACM SIGOPS European Workshop* (Saint-Emilion, France, September 2002). ACM, 2002.