

Autonomously Secure Disks

Kevin Butler, Stephen McLaughlin, Patrick McDaniel, and Youngjae Kim
Pennsylvania State University, University Park, PA

Abstract - *Disks increasingly support multi-boot systems, are accessed remotely, attached to mobile devices, or used in other previously unanticipated operating environments. Because these uses further separate the storage from the data owners, they often lead to new security vulnerabilities. Such concerns call for a change to traditional disk security models. In this paper, we introduce and explore autonomously secure disks (ASD). ASDs enforce security at their interface—thus treating all systems as untrusted entities. Such architectures provide a means to protect user data from rogue or penetrated systems, and significantly reduces the trusted computing base (TCB) for enforcing data security. We introduce three example ASD security policy systems and develop an architecture based on recently introduced hybrid hard drives. Further, we augment the DiskSim simulator with NVRAM services to evaluate the costs associated with managing security contexts within confidentiality and integrity protected secure storage. These experiments show that proper tuning of system parameters can eliminate many of the costs associated with managing security metadata. We conclude by pondering the challenges involved with the integration of ASDs into current operating systems.*

1 Introduction

Storage is rapidly becoming a central conduit for data loss and compromise. Within a mere one-month period between early August and early September 2007, there were 19 reported occurrences of data loss at organizations such as universities, hospitals, financial and health institutions, and government agencies including the military, each caused by improper access, disposal or theft of computer equipment [4]. In each of these cases, sensitive information on hundreds or thousands of people was compromised. For example, at Pfizer, an employee without authorized access retrieved the records of up to 34,000 employees, gaining access to information such as names, Social Security Numbers, and bank and credit card information [32]. In another case, laptops containing the mental health histories of over 300,000 people, and the full names and social security numbers for almost 2,000 people, were stolen from the Pennsylvania Public Welfare Department [44].

Recently, storage manufacturers such as Samsung and Seagate have introduced *hybrid hard disks* and *full disk encryption* devices that augment traditional hard disk mechanisms with additional computational, security, and storage features. This paper considers ways to exploit this and other new disk functionality to address increasing storage security concerns. We designate these security aware devices as *autonomously secure disks* (ASDs). While proposals such as NASD, self-securing storage, and block-based security for network attached disks have considered various methods of providing security semantics and some access control mechanisms, we consider the drive as a completely separate device capable of acting independently of other entities such as metadata servers [3].

An autonomously secure disk (ASD) is a storage device that enforces a security perimeter at its external input/output interface. The ASD is autonomous in that it uses its security policy to govern the hosts and operating systems to which they are connected, i.e., the parent system is *untrusted*. The semantics of the security policy enforced by the ASD is up to the developers of the controller firmware, and bound only by imagination and the computational and storage requirements of its enforcement. However, such security often comes with a cost; one must be cognizant of the ways that enforcement changes performance under the expected workloads.

ASDs change the security model accepted by traditional operating systems. Currently, any OS with access to the disk may access and modify any data contained in it. While such arrangements were acceptable in the past, recent changes to storage architectures and operating environments mandate different security models. For example, multi-boot operating systems are now commonplace. In these systems, every booted operating system may access and arbitrarily modify any partitions of other operating systems. Hence, an operating system stored on the drive is victim to every system vulnerability of *every* OS booted by the system. Further, the increasing ubiquity of interface-level access of remote storage requires careful evaluation of security, e.g., iSCSI [13]. ASDs provide a flexible means of addressing the security requirements of these and a great many other storage environments.

In this paper, we present extended examples of security policy systems implemented using ASDs, and con-

sider architectural requirements of these enhanced disks. We simulate to understand the tradeoffs between disk performance and flash memory storage through the collation of metadata into *integrity sets* that comprise multiple disk sectors. In support of this evaluation, we develop a generic block driver used to drive live workloads to the DiskSim simulator. We augment DiskSim with a flash simulator that implements a detailed model of non-volatile memory. Our extensive evaluations show that security guarantees such as confidentiality and integrity can be implemented in an ASD with surprisingly little overhead; our measurements indicate as little as 2% percent reduction in IOPS in a mail-server workload.

The next section begins by illustrating the use of ASDs in implementing three widely used and highly useful security policy systems. Section 3 describes the architecture and motivating applications for ASDs. Section 4 describes how we emulate ASDs and simulate operation of security features. Section 5 provides an evaluation of performance under a number of differing workloads. Section 6 discusses issues relating to the integration of ASDs into existing systems. Section 7 examines related work, while Section 8 concludes.

2 Autonomously Secure Disks

The following subsections introduce three applications of increasing sophistication to show the utility, flexibility, and challenges of implementing ASDs. We begin in the next subsection by considering how the canonical storage device—a confidentiality and integrity protected disk—can be implemented in an ASD.

2.1 Secure Data Storage

Authenticated and integrity protected disk provide protections for data at rest. Credentials (keys) are either stored in tamper resistant hardware, a secure partition, or provided by a host OS at boot time. The keys are used to encrypt and decrypt data, and to sign¹ and validate the integrity of the data, e.g., as in Seagate’s recently released full disk encryption disks [56]. Because the keys are unavailable to the adversary, physical access to the platters or disk hardware (while the OS which owns a partition is powered down or suspended) provides no ability to either view the plaintext data or modify/corrupt its contents without detection. ASDs are a natural vehicle for implementing these simple yet highly useful devices.

Authenticated Encryption (AE) is a method of encrypting data that provides both confidentiality and integrity.

¹Such “signing” can use either HMACs [39] in symmetric key system or digital signatures [31] in a asymmetric key system.

Several algorithms are defined for AE which include Galois Counter Mode (GCM) [20] and Counter Mode with CBC-MAC (CCM) [19] Both of these modes provide confidentiality through encryption and integrity through the calculation of HMACs, which are stored for future comparison. When a ciphertext record is read, an HMAC is calculated for it and compared to the one stored the last time that record was written, in order to verify the integrity of the record. For a given plaintext, each of these modes produces a ciphertext of the same length and an HMAC of a fixed length, typically 128 bits.

Consider an ASD implementing confidentiality and integrity using AE. The model requires that the ASD completely manage the creation and storage of HMACs used for integrity. A ramification of this is that if the HMACs are stored on the disk platters, they must be outside of the region accessible through the disk interface—thus preventing manipulation by untrusted operating systems or offline access. Further, because the HMAC expands the length of the output beyond that of the input, disks implementing AE cannot store the HMAC sequentially along with the corresponding ciphertext on the disk. While it would be possible to store HMACs elsewhere on the disk, this will lead to additional disk seeks for each block request. Storing the HMACs on the disk also uses additional disk space, limiting the capacity of the disk to hold user/system data.

One option would be to use other storage available to store HMACs, i.e., the NVRAM provided by hybrid drives. The separation of flash from the disk allows for a clean separation of data and security metadata. The flash memory need not be visible to the OS, as it can be managed completely by the disk, and because it is separate from the disk, it can be accessed in parallel with the disk. We consider how the costs and design trade-offs of such a disk architecture in Section 5.

The length of an associated unit of plaintext and therefore ciphertext is an important detail of any implementation of authenticated encryption. If a single disk sector is used as a unit of ciphertext, the space required for storing MACs and IVs is approximately 5.4% of the size of the entire, disk assuming 512 byte sectors. Using IEEE P1619.1 [33], a MAC contains 128 bits of output and requires 96 bit IV. In this case, a 1TB disk would require 54GB of NVRAM to store the MACs. To mitigate this cost, we aggregate disk regions using *integrity sets*—fixed size groups of adjacent sectors for which a single MAC is calculated and stored. When a subject writes one or more blocks in an integrity set, authenticated encryption is performed on the entire set, and a single MAC and IV stored for the set. When a subject reads one or more blocks in a set, authenticated decryption is performed on the whole set. The necessary blocks are extracted from the ciphertext, and a MAC is calculated and compared against the

one stored in NVRAM.

By controlling the size of integrity sets, one can control the amount of space needed in NVRAM for storing MACs and IVs. Of course, this savings in space is not without a cost in time. We have shown that the computational costs of performing the integrity functions in CCM and GCM increases linearly in the number of sectors per integrity set [3]. This implies that the cost of computing a MAC for a set of n sectors is equal to the cost of computing n MACs, one for each sector in the set, plus a constant setup time.

Another advantage of using integrity sets arises from how modern operating systems handle block-level requests. When a block-level request arrives at the I/O scheduling layer, requests for adjacent disk blocks are merged together to reduce the number of requests sent to the disk and therefore disk seeks. This in turn also minimizes the number of MAC calculations as the size of an integrity set in sectors approaches that of the mean number of sectors per request.

2.2 Capability-Based Access Control

Capabilities are access control mechanisms used to govern access at arbitrary levels of granularity [18]. To simplify, a system governs access through unforgeable capability “tokens”, where a user/system/process is permitted access to an object if they can provide the token that is associated with it. Capabilities can be associated with singular objects or aggregates, or even with entire systems [18,58].

As contemplated by the authors of the object based storage (OSD) security architecture [61], capabilities are an appropriate model for governing access in disk systems. The reasoning is that they provide enormous flexibility in implementing access controls at the operating system. The disk will only provide access to systems that can supply the appropriate capabilities. How those capabilities are created, managed, and delegated between systems is entirely up to the systems to which the disks are connected. Thus, issues such as separation, credentials management, and other aspects of the security is outside the scope of the disk. The recently studied Type-Safe Disks [59] explored a particular implementation of capability-based access controls that reflects this model.

Capability based protection state requires two data structures: a) the unforgeable capability token and b) an associated descriptor containing per-object rights. We propose that both of structures be maintained in NVRAM to ensure the secure storage and fast availability of the protection state to the enforcement mechanism. The remainder of this section considers how we would implement this disk system in an ADS with NVRAM.

2.2.1 Capability Representation

The granularity and structure of capabilities has an impact on the amount of space required in NVRAM, the amount of processing power required in the disk’s controller or processor, the complexity of the enforcement mechanism, the degree of coupling with the File System (FS), and the representation of the capability token. We examine these factors for three system artifacts representing different levels of granularity, files, pages, and sectors. The summary of these parameters for the three granularities can be seen in Figure 1. Note that an ASD can implement each these models, even simultaneously, with a single implementation.

Files. The implementation of capabilities at the file level requires the direct involvement of the parent filesystem. In this, the FS carefully tracks the relationships between the disk and filesystem operations, and manages the capabilities based on the associations between files and blocks. Such implementations often require more processor overhead than finer-grained, more autonomous methods, due to the maintaining of data structures that represent file and directory relationships. It is however advantageous in that it requires less space in NVRAM for capabilities than page or sector level implementations that must maintain capabilities for the entire disk, as they lack knowledge of block allocation. File level capabilities also offer a familiar representation of capability tokens in the form of a file descriptor that can be returned by the `open()` system call—thus enabling application level control of capabilities.

Pages. Most modern microprocessor architectures supply page tables that map each process’s virtual address space to pages of physical memory. Pages can be shifted, or *paged*, in and out of main memory into secondary storage, providing processes with address spaces larger than physical memory. Because of this mapping from physical memory pages to those stored on the disk, page table data structures may be used to maintain per-page capabilities. These could be used to control not only how processes access the pages in memory, but how the OS may read and write these pages to disk on behalf of processes. Per-page capabilities are convenient, as most Oses describe block level requests in terms of pages. For example, the Linux `bio` structure, the basic unit of IO requests to block devices, is composed of structures that map to pages of physical memory [16]. All an OS need do is include the correct subject credentials in these requests to obtain per-page capabilities. These capabilities could then be stored in the page table descriptors for each processes and retrieved when their corresponding pages must be paged in or out.

Page granularity capabilities will require more space on

Parameter / Granularity	File	Page	Sector
required NVRAM space	Low	Medium to High	Medium to High
required processing power	High	Low (constant time)	Low (constant time)
coupling with FS	Very High	Simple with VM	None
complexity of enforcement mechanism	High	Simple	Simple
representation of capability token	Per File/File Descriptor	Per Page Stored in page table	Single, per-subject, Possibly a cryptographic key

Figure 1: Comparison of capability granularities.

Subject	Capability Token	Set Bitmap
S1	1e45e77a4c923	R R R W W ...
S2	3ab4d7492731c	RW RW RW RW RW ...
	:	
	:	
SN	3f971639c9283f	0 0 0 0 0 ...

Figure 2: An example of how per-subject capability tokens are represented with an associated capability bitmap. Subject S1 has read access to the first three integrity sets and write access to the next two. Similarly, S2 has read and write access to the first five integrity sets, while SN has no access to any sets.

average than file level, as the capabilities for the whole disk are maintained for each subject regardless of the number of files. However, the amount of required space is static, and once allotted, will not change unless more subjects are added. The only coupling between the disk and the FS/VM subsystems is the requirement that VM pages always align with on-disk pages. Only a small processor overhead is needed to initially distribute the per-page capabilities. Once the subject has obtained the capability, the OS need only forward it to the disk when reading or writing a page on its behalf.

Sectors. Representing per-sector read and write capabilities as a bitmap or capabilities list for each subject is not practical due resource costs. Instead, we propose maintaining capabilities at the granularity of the *integrity sets*. This differs from the proposed page granularity in that it does not depend on the VM subsystem to align pages with the disk. Assuming a large enough set size, capabilities could be maintained in static, per-subject bitmaps without exhausting the space in NVRAM. Such a bitmap would map two bits, enough to express read, write or read and write capabilities, to each integrity set as seen in 2. A subject would obtain a single capability token from the disk which would be used for subsequent disk accesses to map to that subject’s capability bitmap stored in NVRAM.

There are several advantages to this method above the other two. First, because the access control is being done

at the same granularity as the authenticated encryption, a subject’s symmetric key could double as their access token. This allows the same infrastructure used for key management to be implicitly used for capability management. This is advantageous as there has been much exploration in the area of key management [14,27,63]. Second, provided a sophisticated enough on-disk processor, this could also allow access control decisions and authenticated encryption to be done concurrently. Such a pairing of authenticated encryption and access control provides the means to not only protect confidentiality and verify integrity, but to proactively protect integrity by denying unauthorized writes. Third, it requires little processing power. A constant time mapping from the requested sectors in a block request to the corresponding entries in a subject’s bitmap is all that is needed to decide whether the request will be granted.

This method will require more space than the file granularity capabilities, and either more or less than those at page granularity depending on the size of each integrity set compared to the system page size. Due to the static size of the per-subject bitmaps, once the necessary amount of space is allotted in NVRAM, it will not increase until the addition of more subjects to the disk.

2.3 Preserving Information Flow

Information flow policies govern how information may flow between subjects and objects. These policies control where data flows to (confidentiality) and where it flows from (integrity). In the confidentiality preserving Bell and La Padula [6] information flow model, subjects and data observe the *simple* property, which states that no subject can read to a level higher than it is authorized for, and the **-property*, which states that no information may be written at a lower level than the subject is authorized for. This multi-level security model (MLS) ensures that data does not flow to subjects (users) without the proper clearance. The integrity-preserving model proposed by Biba [7], separation of duties as proposed by Clark and Wilson [15], and the Chinese Wall model [11], all ensure that data does not come from low-integrity sources, e.g., minimally or

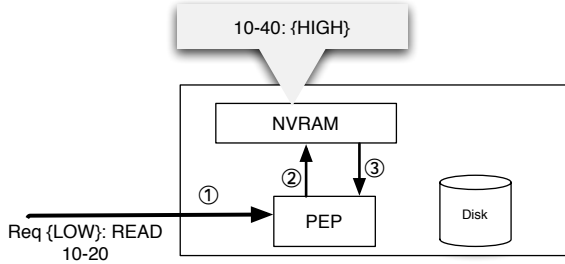


Figure 3: An example of enforcing information flow with disk metadata. A request from a client labeled LOW to read blocks 10-20 is made (step 1). The policy enforcement point, or PEP (usually the processor), consults the non-volatile memory and finds that blocks 10-40 are labeled HIGH (step 2). When this information is returned to the PEP (step 3), it denies the request and no disk access is made.

untrusted sources.

While operating systems such as SELinux, and a small number of applications such as JPMail [28] are increasingly supporting information flow measures, little has been done to protect information flows at the storage layer. ASDs again seem like a natural platform for such protections.

Information flow policy is represented as associations between labels (or label ranges), users, and data. Figure 3 shows an example of ASD/MLS enforcement. A user making a request to the disk has been labeled LOW by the operating system and is attempting to read a set of blocks that has previously been labeled HIGH. Because of the simple security property, a LOW user cannot read HIGH data, so the access is denied at the policy enforcement point – in this case, the processor in the disk mediating the operation. The type of policy enforced by the disk may be modified within the firmware. Alternately, in a high-assurance environment, the manufacturer may burn in custom firmware to the drive such that a specific model must be followed. This approach also has the advantage that labels may be defined *a priori*; with knowledge of these and their semantics, reconciliation of semantics between user labels and those understood by the disk may be easier, as systems can ensure label consistency. However, predefining the label space may also limit the flexibility and granularity of expressible policy.

Regardless of whether labels are predefined or if they can be configurable by the firmware, the drive must be able to understand their semantics. In the previous example, policy enforcement mandates understanding a notion of ordering and comparison: semantics must be in place to determine that because HIGH is a more restrictive level than LOW, a user with level LOW may not be allowed to access data labeled HIGH. This is represented by a *secu-*

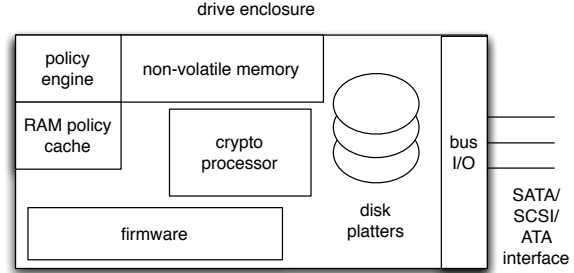


Figure 4: Architecture of an ASD.

urity lattice [6]. Existing research by Hicks et al. has considered policy compliance by examining the intersection of labels in security-typed languages and those derived from an operating system [29]. In particular, the SIESTA tool created by Hicks et al. handles the establishment of connections between the Jif security-typed language and the SELinux operating system. With some modification, a similar tool may be used to provide compliance between operating system labels and those deployed by the storage device. Alternatively, consider a high-assurance system where the entire system enforces information flow throughout and all of the hardware and software to be run is known. In this case, many if not all system parameters may be determined in advance, such that for a given system configuration and policy definition, a hash of the system state may be computed and burned into the drive’s firmware prior to its installation in the system. Then, a system possessing a trusted platform module (TPM) may provide a trusted attestation of the state to the hard disk, which will be able to ascertain the policy as being correct and the system as being trusted to preserve information flow.

3 Architecture

To support the advanced security mechanisms that we have proposed as motivating applications for ASDs, some modifications are necessary to the architecture of currently existing drives. Figure 4 shows what architectural elements are necessary for the deployment of an ASD. Many of these are available in currently-produced hard drives. For example, the Seagate Momentus 5400 PSD [57] and the Samsung HM16HJI [52] are hybrid hard drives containing both magnetic hard disk components and non-volatile flash memory. Both models currently support 256 MB of flash memory. To support encryption, the drive requires on-disk support for cryptographic operations, which is currently available through drives supporting full disk encryption such as the Seagate Momentus 5400 FDE.2 [56], and the Hitachi Travelstar

7K200 [30]. An ASIC within the drive performs cryptographic operations, while primitives such as pseudorandom number generation, encryption algorithms and hash generators are included within the drive’s firmware. These may be leveraged for operations such as authenticated encryption, as described in Section 2.1.

To support more advanced policy structures such as support for capabilities and labels for information flow, mechanisms for policy evaluation and enforcement must be introduced. Policy evaluation at high speeds is already possible in network processors, which quickly evaluate access control lists for incoming packets at speeds of gigabits per second, and general processors are more than capable of servicing requests at the lookup speeds necessary to prevent a bottleneck for the disk. Borders et al. show that with their CPOL policy enforcement architecture, given an AMD Athlon XP2200 processor and 512 MB, request processing could be performed in under 6 microseconds, and often in less than half a microsecond if the policy rule was cached [10]. This amount of overhead is sufficiently small that it essentially disappears given the costs of accessing magnetic, or even flash storage.

Given these considerations, we expect that a likely scenario for deploying policy-based architectural support would include use of an embedded processor such as an XScale PXA270 [34] and RAM for caching policy decisions. Policy frameworks such as security lattices could be burned into ROM at the time of drive manufacture, allowed to be updated through firmware upgrades, or placed in non-volatile memory accessible only to the policy engine, depending on the degree of flexibility required by the drive customer. Even with the reduced processing capability of an embedded processor, the performance overheads of policy-related operations discussed in Section 2 will be minimal relative to disk service times. In addition, many operations can occur in parallel with data retrieval from the drive, thus masking any overheads. With these pieces in place, an ASD could support full-scale policy architectures such as KeyNote [9], STRONGMAN [37], Antigone [41], or OASIS [5].

4 ASD Emulation

Having examined the architecture of an ASD and the applications it supports, we now consider how to evaluate the performance overheads created by the security operations. To fully understand these characteristics, we emulate an ASD by developing extensions to DiskSim [12], which account for the effects of the security operations on disk and flash memory access.

Emulation was chosen because it offers several advantages over other techniques. Storage device emulation requires less effort than implementation, while offering

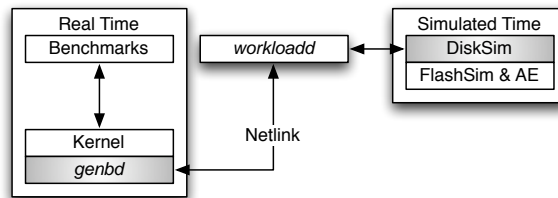


Figure 5: A shim layer is used to interface the kernel driver, *genbd* with the simulated driver layer in DiskSim. Neither the kernel nor DiskSim is aware that there is anything between the gray blocks because *workloadd* translates between real time and simulator time.

comparably accurate results [26]. It also requires less effort to emulate a single component as part of a commodity system than to create a full system simulation. Because the emulated component functions as a part of the commodity system, it can be tested under workloads which more accurately reflect those created by current and widely used systems.

We developed a generic block driver, *genbd*, to move block requests between the kernel and the user-space emulator, *workloadd*, via an asynchronous netlink socket as seen in figure 5. *workloadd* serves two main purposes: it simulates block requests with DiskSim, and satisfies them from a user-space ramdisk. *workloadd* interfaces with DiskSim using an event-based timing loop similar to that used in the Memulator by Griffen et al. [26]. Each simulated event is scheduled to occur at a certain time. Because DiskSim processes requests more quickly than a real disk would, *workloadd* delays simulation of subsequent events until the service time has elapsed in real time. We thus provide consistency between simulated and wallclock time and hence, model I/O overhead more accurately.

4.1 Modeling Security

To simulate security operations and flash memory access, we placed a hook in the DiskSim disk controller code that is called once for each block request. This hook calculates the overhead created by these operations, and adds it to the total time for the disk access. We modeled these operations using the following functions.

- $T(f)$: The time to complete the function f in floating point milliseconds.
- $ISet(o, n)$: The offset and number of sectors for the contiguous integrity sets containing all or part of the block request with offset o and number of sectors n .

- $SpannedSets(o, n)$: The number of integrity sets containing all or part of the block request with offset o and number of sectors n .
- $R_D(s)$; $W_D(s)$: Reads or writes the contiguous set of sectors s to the disk. This operation is simulated by DiskSim.
- $R_F(o, n)$; $W_F(o, n)$: Reads or writes the block request with offset o and number of sectors n . This operation is performed by the flash simulator, as described in section 4.2.
- $E(n)$: Perform authenticated encryption on n contiguous disk sectors as described in section 2.1.
- $R_{ASD}(o, n)$; $W_{ASD}(o, n)$: Reads or writes the block request with offset o and number of sectors n to an ASD. This operation is a composition of the above functions.

We can thus model completion times with the following equations. The time for a read to the ASD is

$$T(R_{ASD}(o, n)) = T(R_D(ISet(o, n))) + T(E(n)) \\ + T(R_F(o, SpannedSets(o, n)))$$

The corresponding time for a write can be modeled as

$$T(W_{ASD}(o, n)) = T(W_D(o, n)) + T(E(n)) \\ + T(W_F(o, SpannedSets(o, n))) \\ + T(R_{ASD}(o, n))$$

Note that writes include the time for a read to the ASD over the set of specified blocks. When a write occurs, the HMAC of the integrity sets corresponding to the modified blocks must be recalculated and the new result stored to reflect the changes made.

4.2 Flash Memory Emulator

To support the experiments detailed in the following sections, we designed and implemented a simple flash memory disk emulator. Integrated into DiskSim [12], the resulting driver is comparable in behavior and operation to SanDisk’s SSD Solid-State Drive and BiMICRO’s E-Disks [8, 54]. A flash memory based solid-state disk operates substantially similar to the that of a similar to conventional block block device/hard drive, except that the storage media is NVRAM.

Our emulator is composed of three software components (an IO device driver, an address mapping module, and a flash core engine). All requests are queued by IO device driver and issued to the flash memory in order.

We use the Flash Translation Layer (FTL) [38] for address mapping. The emulator supports *read*, *program* and *erase* operations corresponding to conventional disk IO read and write operations with two level mapping tables. Every request is serialized over a single memory channel (as in current flash drives). Interested readers are directed to the relevant literature for details on these operations [22, 45]. The emulated flash memory models a large block NAND flash memory [53] containing 2KB pages containing 4 512 byte sectors. A block consists of 64 pages. Read and program operations are performed in the unit of page. The erase operation is performed in the unit of block. We use Kang et al.’s performance measurements to model each NVRAM operation, i.e., 0.027320 us per page read, 0.196370 us per page write, and 1.5 ms for per block erase.

Each page must be erased before it is reused. Thus, a garbage collector is needed to select an appropriate block for erasure when no “fresh” page is available. When the garbage collector is called, a candidate block is selected based on the ratio of the number of invalid pages to valid pages. As highlighted in the next section, the need for erasure on certain writes can induce significant variance in the write delay.

4.3 Experimental Setup

All tests described in the following section were performed on a 1.86 GHz Intel Core2 CPU with 1GB of RAM, running Ubuntu Linux with a 2.6.20-16-generic kernel. The filesystem used for the tests was ext2, and used the Anticipatory I/O Scheduler. Note that no actual disk was used, as all block requests were satisfied from the emulator’s user-space ramdisk (pinned to physical memory using `mlock()`). We provided a parameter of 512 MB of flash memory to the flash simulator.

The maximum size of each block request was set to 255 sectors, the Linux kernel default, and the maximum number of outstanding requests was set to 32, which is the maximum number of outstanding commands in both the ATA Tagged Command Queueing and Native Command Queueing [17] standards, as well as the upper limit that could be efficiently queued in user-space. To simulate disk requests we used the DiskSim 3.0 simulation environment. The chosen disk model was the default Cheetah 4LP, which is a 4.5 GB, 10,000 RPM SCSI drive with a 512 MB cache.

We automated the benchmarking process using the Auto-pilot benchmarking suite [64], which we configured to run each test a minimum of 20 times, and to compute 95% confidence intervals for the mean elapsed, system and user times using Student’s t-distribution. In between each run the device created by `genbd` was unmounted to ensure a cold cache.

Two benchmarks were used, PostMark version 1.51 [35] and a simple in house benchmark. PostMark creates workloads similar to those of email servers. It performs transactions on a set of many small files. A PostMark transaction consists of either a read or a write and either a creation or a deletion. We configured PostMark to perform 50,000 transactions on 20,000 files ranging in size from 500B to 20KB, using buffered I/O. Our in house benchmark does repeated sequential writes and reads of the entire disk, in order to test the effects of large sequential accesses on flash memory. We configured it to perform 2 write and read pairs on the entire disk.

As discussed in Section 3, we assume a custom on-disk ASIC for performing authenticated encryption operations. Both CCM and GCM modes require 128-bit block ciphers. We assume an AES-128 block cipher, which we simulated based on measurements from AES implementations on ASICs [49]. To simulate authenticated encryption on n 512 byte sectors of data, we calculate the time needed to perform $n \times 32$ encryptions using an AES-128 cipher, 32 being the number of 128 bit blocks in each 512 Byte sector.

5 Evaluation

ASDs must manage more meta-information than traditional disks. Hence, an essential issue is cost; how much does the added function impede the normal functioning of storage. Whether it be due to managing HMACS, capabilities, labels, or any other security context, there will be trade-offs between the security provided, resource usage, and run-time performance. A prerequisite of an understanding of the feasibility and utility of ASDs is an understanding of this cost. The following study begins to map this space by evaluating the canonical security policy, confidentiality and integrity-guaranteed storage, in our emulated disk system.

The key trade-off parameter of the secure disk is integrity disk set size. More specifically, the set size determines the trade-off between performance and flash memory requirements. In the following experiments, we observe the overheads caused by security operations and flash memory access, as well as disk level statistics which hint at possible methods for optimization. An integrity set size of zero is the baseline case in which no security functions are executed (and no flash memory is accessed).

5.1 Small Random Access Workload

The completion times for the PostMark tests are shown in figure 6. They increase with an average slope of 0.032 seconds overhead per additional sector in the integrity set size. Similarly, the IOPS for these test are seen in figure 7.

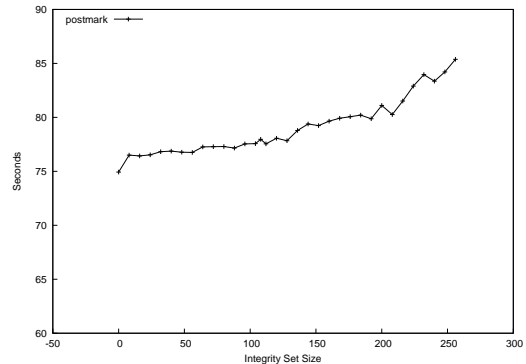


Figure 6: Average completion times for various integrity set sizes

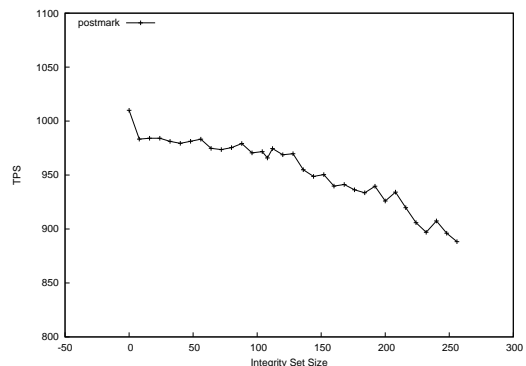


Figure 7: Average IOPS for various integrity set sizes

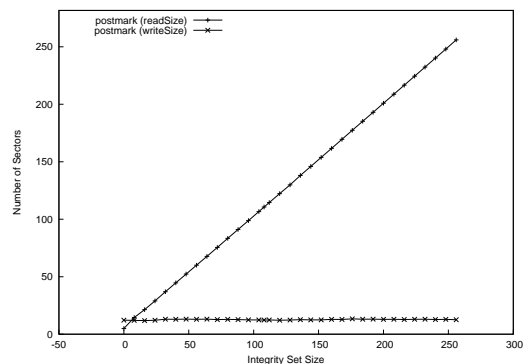


Figure 8: Average block request size vs integrity set size

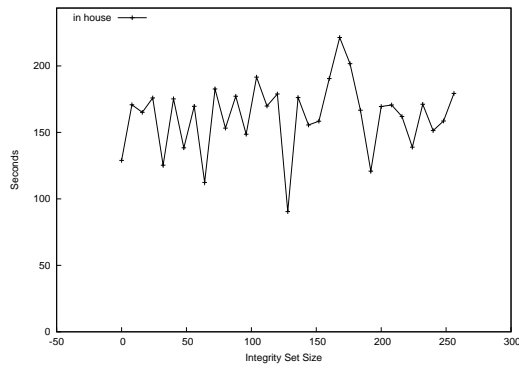


Figure 9: Completion times for the in house benchmark

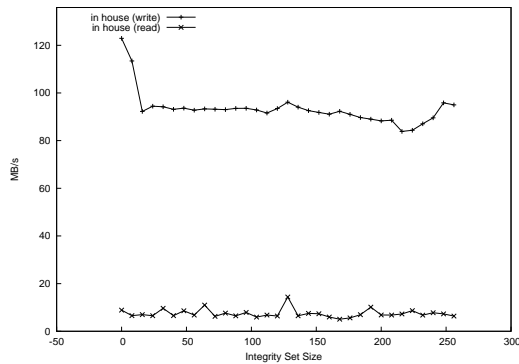


Figure 10: Read and Write throughputs for the in house benchmark

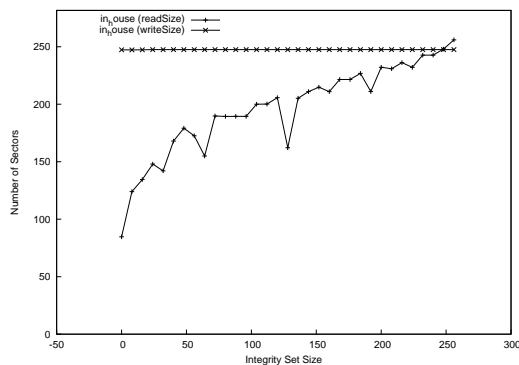


Figure 11: Average request size for the in house benchmark

They decrease with integrity set size at an average rate of 0.4 IOPS per sector. In both graphs, there is a slope much steeper than the average between the points with set size zero and eight, due to the introduction of flash access to the baseline.

This steady decrease in performance is due to the steady increase in modified request sizes as seen in figure 8. The rate of decrease in performance is considerably smaller than the rate of increase in read request size for several reasons. First, as integrity sets become larger, fewer reads and writes are done to flash memory. Writes to flash can be especially costly when an erase operation must be done part of garbage collection. Second, integrity set size mainly affects transfer time, which is closely related to request size, while seek time, a more significant contributor to access latency, is more closely related to request contiguity. We also saw some higher than normal disk cache hits rates due to the fact that we could only use 512 MB of the disk because of size limitations of the ramdisk.

In general, for workloads that perform small random transactions, disk performance is inversely proportional to integrity set size by a small constant. This is due to increased read request sizes, which increase transfer time. Some of the affects of the increased sizes are absorbed by a reduction in the number of flash memory accesses.

5.2 Large Contiguous Access Workload

The completion times of the in house benchmarks are shown in figure 9. Unlike the random workload, in which performance degraded predictably with integrity set size, completion times oscillate with respect to set size. Note that the shortest completion time in this case is not at set size zero as one might predict, but instead at set size 128. Similar but smaller oscillations are seen in the read and write throughputs as shown in figure 10. Note that the set size with the highest read throughput is also 128.

Demonstrated in Figure 11, the same set sizes which have the shorter completion times and higher throughputs also have smaller average request sizes after modification. The reason for the smaller modified requests is better alignment of particular set sizes with request size before request modification. If a block request is slightly larger than an integrity set, the entire neighboring set will be read and it's metadata will be updated in flash memory. This is why set sizes which perform well are next to sizes that perform poorly. The good performers have set size slightly less than or equal to the average block request size before modification. Schindler et al. observed similar erratic performance variances (whose root cause was also similar) when aligning I/O accesses with disk tracks [55].

The above results show that unlike the random workload, there is no approximately linear relationship between integrity set size and performance for a contiguous

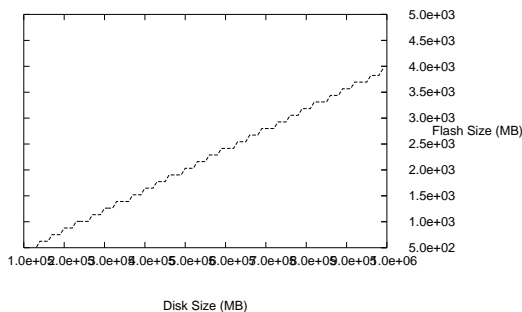


Figure 12: The contour line for best completion time of 75.5s shows the trade-offs between flash memory size and disk size.

workload with large reads and writes. An important ramifications of this is that performance is not proportional to disk size. This is because disk combined with available flash memory size dictates integrity set size. This is advantageous in that the optimal integrity set size may be chosen for a random workload, and I/O system parameters may be tuned to align requests with the chosen set size.

While contiguous workloads with large block request sizes performance is independent of disk size, it was shown above that for random workloads, disk performance is inversely proportional to integrity set size. This raises the question of the feasibility of this scheme for disks much larger than the one used in these simulations. As commodity 1TB disks are now available, we wish to use the results shown here to make predictions about how the trade off between NVRAM size and disk performance.

5.3 Large Disk Analysis

We now use the simulation results presented in the preceding section to model the time/space trade offs in large disks. Our model is developed by extracting the salient effect of disk parameters from the emulated environment using linear regression, and projecting that model on more modern disks. Specifically, this model allows us to understand the performance impact of pairs of Disk size and flash memory size. These two parameters are mapped onto IOPS and completion time by the following function:

$$Performance(D, F) = MinSetSize(D, F) \times m + b$$

Where D is the size of the disk in MB and F is the

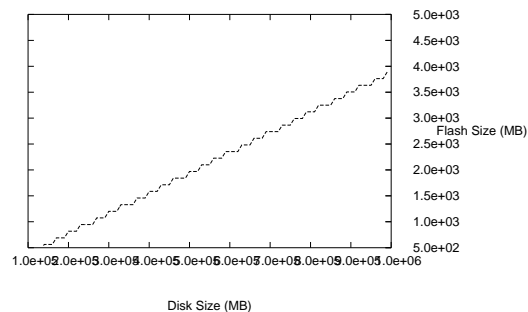


Figure 13: The contour plot for 995 IOPS is the same as that for completion time because the same integrity set size is optimal for both.

size of the flash memory in megabytes. The function $MinSetSize(D, F)$ finds the minimum integrity set size needed to hold all metadata for a disk of size D MB in F MB of flash memory. m is the slope obtained from linear regression, and b is the intercept. The function $Performance$ can compute either IOPS or completion time. Because $Performance$ is a function of two variables, we examine contour plot of the level with highest performance.

The contour lines for completion times are shown in figure 12, and IOPS in figure 13. The two plots are almost identical because for both the integrity set size of 16 was used, which is the smallest set size that could store all security metadata for the ranges of flash and disk sizes. In order to achieve optimal performance in a 1TB disk, 4GB of flash memory are required. To achieve the same performance on a 4.5GB SCSI drive requires only 156 MB of flash memory. Comparable performance with a 2% decrease in IOPS can be obtained by using the next largest integrity set size of 32.

6 Discussion

In this section, we consider some of the issues that arise with the operation of ASDs.

6.1 RAID

ASDs will be deployable in RAID configurations with minimal to no additional operations required by a RAID controller. Because of metadata is kept on the ASD, none

of this information needs to be exposed to the RAID controller, which can continue to make block requests to the drives. Configurations such as RAID-0 (striping) or RAID-1 (mirroring) are relatively straightforward, as security metadata associated with the on-disk data will be stored in the non-volatile memory associated with the respective disk.

A RAID-5 configuration may be more complex as parity information can be used to reconstruct a disk. This could have ramifications if the ASD, for example, must preserve information flow with different levels of security on the drive. In this case, a conservative approach would be to label parity information at the highest level of data written to the drive, or the highest level supported by the system. The former case would present a challenge if subsequent higher data is written to the RAID array; if this was the case, then all parity metadata would have to be relabeled at the higher level. However, this approach prevents the arbitrary escalation of information as would be necessitated in the latter model. Parameterizing these features is a policy decision that may be best answered by storage administrators depending on their needs.

6.2 OS Modifications

To support ASDs that implement policy enforcement functions, such as access control decisions, the OS requires modifications to handle access denial messages from the disk. The hardware driver must be modified to recognize these messages and propagate them up to the OS; the message to be returned to users will be implementation-specific. Drivers must also be cognizant of metadata sent to the disk layer by the operating system, such as labels. There may, however, be support within the OS structures to set this data; for example, the `bio` structure in Linux, which packages I/O requests to the block layer, has fields that may contain user-defined information, such that metadata may be passed from the OS to the hardware level. The `ioctl` control commands may also be used to directly push information to the disk, though cleaner interfaces would be preferable. We defer further consideration of these issues for future exploration.

6.3 Disk Backup

Backing up ASDs is largely outside the scope of our discussion; however, to ensure metadata backup would require interfaces with the backup application and facilities to transfer this information to the operating system. In particular, given the information flow-preserving disk example, the OS would need to possess a privilege level equal to the highest level of security label stored on the disk. Moreover, for the backup application to preserve information flow, it should be optimally written in

a security-typed language such as Jif, which will respect the security labels associated with the data to be backed up. Such a secure backup system requires cognizance and enforcement of security labels through both the application and operating system, requiring the OS modifications discussed above as a baseline. We defer consideration of fully integrating a system-wide information flow preserving architecture for future work, but note that with these disks, it is possible to implement such a system.

7 Related Work

Securing data below the level of the file system has been an area of focus, particularly with the advent of network-attached disks that accept direct block reads and writes. Network-attached secure disks (NASD) [23, 24] sought to replace the block model with *objects* of variable size that could provide greater semantics for data. The metadata associated with these objects would then be stored on a metadata server. hashed and encrypted blocks and groups of blocks called objects, storing the metadata on a server. Similarly, SNAD [43] uses keyed hashes extensively, calculating either a digital signature or HMAC value over blocks. The latter scheme is similar in execution to ours, but relies on the client to perform the cryptographic operations and store HMACs, rather than the disk, and does not take into account the ability to amortize costs and storage over multiple blocks. SCARED [51] provides data integrity but not at the block layer, so operations cannot be performed by the disk. SNARE [65] shares some similarities to NASD and SNAD but relies on capabilities and is best suited for a remote storage system. While our proposed capabilities-based application shares similarities with the block-based capabilities work by Aguilera et al. [2], that proposal relies on access control at a metadata server. By contrast, we consider enforcement directly within the disk itself. In addition, none of these schemes consider authenticated encryption using modes specified by the IEEE P1619.1 standard.

The concept of storage that is independently secured was explored by Strunk et al. [60]. In this model, the focus is on objects that act as capabilities, in a similar manner to NASD but security additions focus on intrusion detection [48] and recovery from these types of attacks, through the use of on-disk audit logs and considering primarily versioning of objects. Our proposal differs in that both object storage and enforcement of policies is performed within the disk, forming a smaller security perimeter.

The Venti storage system [50] is an archival system that relies on write-once access. Their system considers block level performance, and attempts to optimize storage by seeking opportunities to compress blocks before adding them to the archive. Vilayannur et al. [62] also considered

optimizing performance through tuning of parameters, but make characterizations and modifications to parallel file systems. By contrast, our approach works at the block layer and is thus largely independent of the file system running above it.

Oprea et al. [47] consider an on-disk model for protecting block integrity using calculated entropy. Their assumption is of an untrusted disk where the client performs all calculations and leverage the block's entropy to make decisions whether to hash the blocks or not. This scheme yields large reductions in required storage but requires clients to retrieve their own integrity information. This concept has also been incorporated into cryptographic file systems [46], where file in conjunction with the construction of a Merkle hash tree [42] to represent the file contents; this approach of building hash trees for integrity has also been considered in systems such as Cepheus [21], Farsite [1], SUNDR [40], and Plutus [36], while SiRiUS [25] has hash tree validation at the file level. For a given file, only blocks considered high entropy are hashed, as blocks with low integrity are considered already structured; it is assumed that only if the block is tampered with will its entropy increase as it becomes corrupt, such that the entropy measurement takes the place of any hashing. This approach requires an amount of trusted storage accessible for each file. Our proposals differ in that we are concerned solely with blocks at the disk level, and all of our constructions and operations are meant to be able to be implemented within the disk itself.

8 Conclusion

Autonomously secure disks (ASDs) present a a new paradigm for securing storage through enforcing security at their interface, thus treating all systems as untrusted entities and significantly reducing the trusted computing base. We examined three potential applications of ASDs, including capability-based access control and information flow preservation. We developed an emulator to understand the performance characteristics of ASDs, driving live workloads to Disksim and providing extensions to account for flash memory access as well as magnetic storage. Our evaluation shows that by tuning system parameters, much of the overhead of managing security metadata can be mitigated. Future work will consider further integration of ASDs into operating systems and exploring new potential policy architectures that they engender.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch,

M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

- [2] M. K. Aguilera, M. Ji, M. Lillibridge, J. McCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-Level Security for Network-Attached Disks. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, Apr. 2003.
- [3] Anonymous. Work anonymized for submission.
- [4] attrition.org. Entities That Have Suffered Large Personal Data Loss Incidents, Sept. 2007. <http://attrition.org/dataloss/#2007>.
- [5] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security*, 5(4):492–540, November 2002.
- [6] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corporation, Bedford, MA, 1973.
- [7] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, Apr. 1977.
- [8] Bitmicro. <http://www.bitmicro.com>.
- [9] M. Blaze, J. Feigenbaum, and A. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Proceedings of the 1998 Cambridge Security Protocols International Workshop*, pages 59–63, Cambridge, UK, May 1999. Springer-Verlag Lecture Notes in Computer Science.
- [10] K. Borders, X. Zhao, and A. Prakash. CPOL: High-Performance Policy Evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, Alexandria, VA, USA, Nov. 2005.
- [11] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, Apr. 1989.
- [12] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, Jan. 2003.

- [13] S. Chaitanya, K. Butler, P. McDaniel, A. Sivasubramaniam, and M. Vilayannur. Design, Implementation and Evaluation of Security in iSCSI-based Network Storage Systems. In *Proceedings of 2nd International Workshop on Storage Security and Survivability (StorageSS 2006)*, Alexandria, VA, Oct. 2006.
- [14] P. Chen, J. Garay, A. Herzberg, and H. Krawczyk. Design and Implementation of Modular Key Management Protocol and IP Secure Tunnel. In *Proceedings of 5th USENIX UNIX Security Symposium*, pages 41–54. USENIX Association, 1996. Salt Lake City, Utah.
- [15] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, Apr. 1987.
- [16] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O’Reilly, third edition, 2005.
- [17] B. Dees. Native Command Queuing - Advanced Performance in Desktop Storage. *Potentials, IEEE*, 24(4):4–7, Oct. 2005.
- [18] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.
- [19] M. Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, May 2004. NIST Special Publication 800-38C.
- [20] M. Dworkin. Recommendation for Block Cipher Modes of Operation: The Galois/Counter Mode (GCM) for Confidentiality and Authentication, May 2006. NIST Special Publication 800-38D (DRAFT).
- [21] K. Fu. Group Sharing and Random Access in Cryptographic Storage File Systems. Master’s thesis, MIT, June 1999.
- [22] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, June 2005.
- [23] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, USA, Oct. 1998.
- [24] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A case for network-attached secure disks. Technical Report CMU-CS-96-142, Carnegie Mellon University, Pittsburgh, PA, USA, Sept. 1996.
- [25] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the 10th ISOC Symposium on Network and Distributed Systems (NDSS’03)*, San Diego, CA, USA, Feb. 2003.
- [26] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger. Timing-accurate storage emulation. In *FAST*, pages 75–88, 2002.
- [27] H. Harney, U. Meth, A. Colgrove, and G. Gross. GSAKMP: Group Secure Association Key Management Protocol, June 2006. RFC 4535.
- [28] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, pages 153–164, Miami, FL, Dec. 2006.
- [29] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. Integration of selinux and security-typed languages. In *Proceedings of the 2007 Security-Enhanced Linux Workshop*, Baltimore, MD, March 2007.
- [30] Hitachi. Hitachi Ultrastar 15K300 3.5-inch Enterprise Hard Disk Drives. [http://www.hitachigst.com/tech/techlib.nsf/techdocs/F0954C336AF5E24F862572C200563CB3/\\$file/Ultrastar_15K300_final_DS.pdf](http://www.hitachigst.com/tech/techlib.nsf/techdocs/F0954C336AF5E24F862572C200563CB3/$file/Ultrastar_15K300_final_DS.pdf), Apr. 2007.
- [31] R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *Internet Engineering Task Force*, April 2002.
- [32] L. Howard. Pfizer Has Another Breach of Security. New London Day, Aug. 2007. <http://www.theday.com/re.aspx?re=7b6d810e-f7ef-4243-a86c-bfdb9093f983>.
- [33] IEEE. IEEE P1619.1/20 Draft Standard for Authenticated Encryption with Length Expansion for Storage Devices. <http://attachments.wetpaintserv.us/2Qjro\%24n0iiv7kYZoz4BTmw\%3D\%3D326044>, June 2007.

- [34] Intel. Intel PXA270 Processor for Embedded Computing, Sept. 2007. <http://www.intel.com/design/embeddedpca/applicationsprocessors/302302.htm>.
- [35] K. J. PostMark, A New Filesystem Benchmark, 1997.
- [36] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, Apr. 2003.
- [37] A. D. Keromytis, S. Ioannidis, M. D. Greenwald, and J. M. Smith. The STRONGMAN Architecture. In *Proceedings of the Third DARPA Information Survivability Conference and Exposition (DISCEX III)*, Washington, DC, Apr. 2003.
- [38] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.
- [39] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Apr. 1997.
- [40] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, San Francisco, CA, Dec. 2004.
- [41] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114, August 1999. Washington, DC.
- [42] R. Merkle. Protocols for Public key Cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*, pages 122–133. IEEE, April 1980. Oakland, CA.
- [43] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong Security for Network-Attached Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, Jan. 2002.
- [44] J. Murphy. Computers stolen from welfare office. Harrisburg Patriot News, Sep. 11 2007. <http://www.pennlive.com/midstate/patriotnews/article121468.ece>.
- [45] H. Nijjima. Design of a Solid-State File Using Flash EEPROM. *IBM Journal of Research and Development*, 39(5), 1995.
- [46] A. Oprea and M. K. Reiter. Integrity Checking in Cryptographic File Systems with Constant Trusted Storage. In *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, USA, Aug. 2007.
- [47] A. Oprea, M. K. Reiter, and K. Yang. Space-Efficient Block Storage Integrity. In *Proceedings of the 12th ISOC Symposium on Network and Distributed Systems Security (NDSS'05)*, San Diego, CA, USA, Feb. 2005.
- [48] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger. Storage-based Intrusion Detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, USA, Aug. 2003.
- [49] N. Pramstaller, S. Mangard, S. Dominikus, and J. Wolkerstorfer. Efficient aes implementations on asics and fpgas. In *AES Conference*, pages 98–112, 2004.
- [50] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, Jan. 2002.
- [51] B. C. Reed, M. A. Smith, and D. Diklic. Security Considerations When Designing a Distributed File System Using Object Storage Devices. In *Proceedings of the 1st IEEE Security in Storage Workshop (SISW'02)*, Greenbelt, MD, USA, Dec. 2002.
- [52] Samsung. SAMSUNG's Digital World - Hybride-FlashON HM16HJI, Sept. 2007. <http://www.samsung.com/fr/products/harddiskdrive/hybrideflashon/hm16hji.asp>.
- [53] Samsung electronics. http://www.samsung.com/global/business/semiconductor/productList.do?fmly_id=159.
- [54] Sandisk. <http://www.sandisk.com>.
- [55] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST*, pages 259–274, 2002.
- [56] Seagate. Seagate Technology - Momentum 5400 FDE.2 Hard Drives, Sept. 2007. http://www.seagate.com/www/en-us/products/laptops/momentum/momentum_5400_fde.2/.

- [57] Seagate. Seagate Technology - Momentus 5400 PSD Hybrid Hard Drives, Sept. 2007. http://www.seagate.com/www/en-us/products/laptops/momentus/momentus_5400_psd_hybrid/.
- [58] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP*, 1999.
- [59] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *OSDI*, 2006.
- [60] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*, San Diego, CA, USA, Oct. 2000.
- [61] T10. Information technology - SCSI Object-Based Storage Device Commands (OSD). Project T10/1355D, Revision 10, July 2004.
- [62] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing Tunable Consistency for a Parallel File Store. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*, San Francisco, CA, USA, Dec. 2003.
- [63] D. M. Wallner, E. J. Harder, and R. C. Agee. Key Management for Multicast: Issues and Architectures. *Internet Engineering Task Force*, June 1999. RFC 2627.
- [64] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *USENIX Annual Technical Conference, FREENIX Track*, pages 175–188, 2005.
- [65] Y. Zhu and Y. Hu. SNARE: A Strong Security System for Network-Attached Storage. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, Oct. 2003.