Grains of SANs: Building Storage Area Networks from Memory Spots

Lisa Johansen, Kevin Butler, William Enck, Patrick Traynor, Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
Computer Science and Engineering, Pennsylvania State University
{johansen,butler,enck,traynor,mcdaniel}@cse.psu.edu

Abstract

Emerging storage platforms, such as HP's memory spot, are increasingly becoming smaller, faster and less expensive. Whether intended for holding digital media or personal documents, such systems currently function as independent receptacles of data. As we demonstrate in this paper, however, the true power of these devices is their ability to form the flexible building blocks of larger logical storage systems. Such systems allow for the creation of continuously reconfigurable storage devices, capable of the dynamic addition, removal and repurposing of component nodes. Moreover, such changes can be made transparent to the user's view of the storage system. To illustrate these properties, we design and implement a granular storage system based on memory spots. In so doing, we identify and address significant challenges in areas including organization, security and reliability. We then conduct an extensive analysis of performance and demonstrate the ability to achieve throughputs of greater than 3 Mbps to our unoptimized logical storage device. These results demonstrate the potential for new applications and systems built on granular storage.

1 Introduction

Suppose you could rethink storage. Imagine for a minute that storage was a physical substance like grains of sand that could be placed, relocated, subdivided or combined. The more storage one had, the more storage would be available for whatever purpose was immediate. Movement would be effortless–systems would recognize storage as it became available and use it readily. In this world, storage would no longer be an artifact of the fixed computing infrastructure in your office or home or static token one carries in briefcase, but a highly flexible, reusable, and inexpensive commodity.

What would such a reality enable? Systems could easily and immediately expand or reduce their storage. For example, could I take storage away from my laptop while it is running to allow my television to record an extra hour of shows and return it later? Could I achieve true storage mobility by keeping a terabyte of storage sealed in the bottom of my travel coffee mug I take to work each day? Could GRID system operators arbitrarily redistribute storage allocation by physically relocating portions of grains?

Such a storage substance exists. HP's recently announced memory spot devices that export a simple storage interface over a wireless media [12, 8, 18]. Built on a platform similar to passive RFIDs, these $2mm^2$ devices support up to 4Mb of storage that can be accessed at transmission rates upwards of 10Mbps. With projected costs of between 5 cents and a dollar (US), these *storage grains*[1] are perfectly suited to meet our vision of *granular storage*.

In this paper, we consider the requirements, design and behavior of storage area networks (SANs) built on granular storage. We begin by considering a number of motivating applications and explore the properties of granular storage that separate it from traditional storage systems. A high-level architectural design is given and key research challenges considered. An implementation of our prototype granular virtual storage is described and development experiences detailed. We then present an in-depth study of the operational parameters of a granular storage system. The analysis of our prototype storage system shows that we can achieve high throughput under a range of realistic configurations and workloads. We conclude by commenting on the future of granular storage systems and key remaining technical challenges.

Granular storage is very different from past storage systems. The previously unseen levels of potential access parallelism, transient nature of grains, and physical geography of a granular storage system introduce a number of technical opportunities and challenges. These challenges/opportunities are embodied in the three main thrusts of this work:

[1] Because only incomplete specifications of HP's memory spots are available to the public at the time of writing, we use the term "storage grains" to describe a platform inspired by but not necessarily beholden to their device.

- **Organization** - Data placement–the selection of the specific storage grains on which data are to be located–is a central determinant of the performance and reliability of the system. For this reason, exploring the behavior of allocation strategies is therefore critical.
- **Security** - Storage grains use an unprotected wireless interface and possess very few native security capabilities. These characteristics induce a number of serious threats that must be addressed.
- **Reliability** - Storage grains are expected to be added and removed frequently. The integrity of the storage content following these events must be maintained.

The remainder of this paper details how we address these key issues.

One may be tempted to see shades of granular storage in recently proliferating USB "thumb drives" or other low-cost storage devices. However, such comparisons are misplaced: these devices are essentially convenient removable hard drives. By contrast, granular systems seek to gracefully absorb and release very small portions of the larger logical storage system in a reliable, secure way as the physical environment changes. We begin our analysis of these systems in the following section by considering the vision and motivation for granular storage.

2 Application Framework

Consider the versatility of storage systems built on storage grains. Before embarking on a long trip, an automobile owner attempts to load a new set of maps into the onboard navigation system; however, the storage capacity of this system has been reached. Undeterred, the owner takes a handful of unused storage grains from his laptop and adds them to the vehicle. On reaching his destination, a subset of the storage grains borrowed by the navigation system are moved to his digital camera to enable extended use. Finally, after arriving home, our traveler returns the storage grains to his laptop in order to store his processed photographs.

The creation of a storage infrastructure based on memory spot technology provides a framework for a diverse set of additional applications. Instead of requiring users to transport general purpose computing platforms (e.g. laptops) between locations, they could instead simply dock their portable SAN at any open terminal. If device mounting occurs at boot time, a briefcase embedded with storage grains can carry not only a user's data, but also their home environment and operating system. Because of their small size and lack of moving parts, the use of storage grain-based SANs have great potential in highly dynamic environments. For example, such devices can be embedded within clothing and helmets, providing unobtrusive physical protection for battlefield storage systems and data. The applications of such a flexible logical storage device are unbounded. In order to understand how such applications can be realized, it is necessary to examine the characteristics of a storage system built on these devices.

Reconfiguration is an extremely coarse-grained operation in current storage systems. When capacity is reached, for instance, the solution typically involves device duplication (e.g., adding new disks) regardless of the additional space required. Such over-provisioning is not only cost-ineffective, but also potentially causes a significant under-utilization of resources. For example, in a setting where three devices each require an additional 10MB of storage space, the purchase of three additional hard drives is extremely inefficient. While USB flash memory offers a more reasonable alternative in this regard, the inability to simultaneously divide a single device between multiple machines limits their applicability. Ideally, excess capacity from one device should be relocated in arbitrarily small quantities. Because of their modest individual capacities, storage grains satisfy this objective. As demonstrated in our motivating example, a handful of storage grains could be transparently added to the SAN of each of the previously mentioned devices and, as usage patterns change, be arbitrarily reconfigured and redistributed between them.

Portability and capability in storage systems have traditionally been inversely proportional. Large disk banks, capable of extremely high access rates and storage capacities, are functionally immobile. Conversely, while USB flash storage devices offer increased mobility, they lack high individual throughput and the ability to operate in parallel. While RFIDs have a similar form factor to storage grains and the potential to be used in parallel, the absence of a malleable computation infrastructure and the inability to rewrite content restricts their effective use as the building block of a SAN. However, the ability to organize a significant number of very small storage grains into a logical storage device approximates many the capabilities of larger static systems without sacrificing portability.

In order to create systems based on this architecture, it is necessary to examine the requirements that storage grain-based SANs must achieve. Like all storage systems, the ability to reliably retrieve data is of paramount importance. Because of the small size and potential for damage to individual devices, fault tolerance is of particular concern. Performance must then be addressed in order to understand the behavior of such an infrastructure. Finally, because of the potentially exposed nature of such systems, security must be built in to the initial architecture. Accordingly, the remainder of this paper investigates how each of the above design requirements can be achieved in such a system.

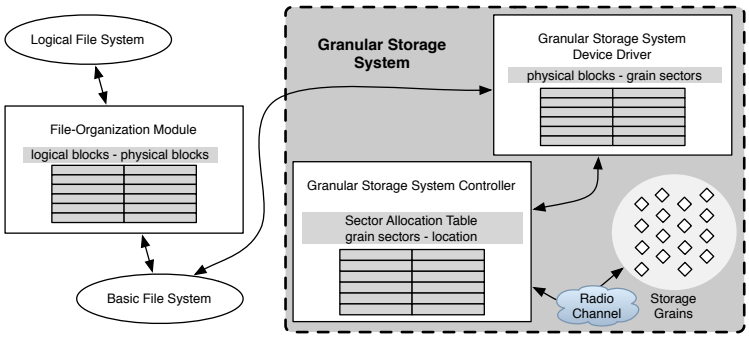

Figure 1: Storage Grain System Architecture

3 System Design

The foundation of our system is based on two central elements: a host machine and a set of storage grains. The host communicates wirelessly with nearby storage grains and is responsible for organizing them into a logical storage device. As is consistent with traditional storage architectures, the host's management makes the existence of individual storage grains transparent to the file system.

This conceptual model of the storage system, shown in Figure 1, exhibits design constraints that motivate three central technological challenges: organization, reliability and security. The methods and strategies by which data is organized affects contention for resources, parallelism, and overall performance. Security provides the protection of the confidentiality and integrity of data. This is particularly challenging in a granular storage system because both the physical devices and the communications between them are vulnerable to attack. Reliability, which provides the capacity to recover from error and loss, is also necessary because of a high failure rate caused by grain loss, failure, theft or physical reassignment.

This section introduces the low level architecture of a storage grain-based SAN. We begin by discussing the components of the system. We then examine the impact of the design decisions made for each element on the organization, security and reliability of the overall system.

3.1 Architecture

Figure 2 illustrates the major functional components of a storage grain-based SAN. The process of accessing data stored on a SAN occurs when an application makes a system call requesting read or write access to a file. The call is intercepted by the file system in the kernel and passed as a block request to the storage grain device driver. The device driver then requests specific sectors from the storage grain controller, which communicates directly with the physical storage grains. The results of a successful request are then returned to the user via the reverse path.

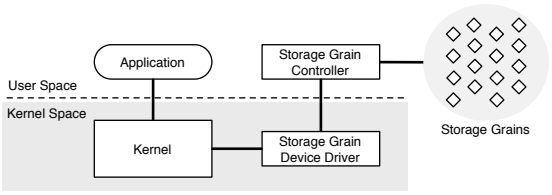

Figure 2: Functional components of a storage grain-based SAN.

The following subsections discuss the design of the device driver, storage grain controller and storage grains. We discuss the design tradeoffs made at each phase and their potential impact on the operation and performance of the system.

3.1.1 Device Driver

The device driver is a loadable Linux kernel module implementing the block interface. Upon receiving requests from the kernel, the device driver translates requests for block addresses into sectors. Requests are then sent to a user-space storage grain controller via a netlink socket. Responses from the storage grain controller are converted into block request responses and returned to the user space application via the kernel. As currently implemented, this device driver is capable of supporting exactly one storage grain controller.

Designing a device driver with minimal capabilities serves a number of purposes. First, it minimizes the addition of complex functions operating in kernel-space. Moreover, implementing the majority of functional elements in user space permits the maximum flexibility and support for the remaining pieces of the prototype system (e.g. debugging, simplified interfaces, portability). For improvements in performance, many of the intelligent functions are likely to eventually migrate into the kernel as this codebase matures.

3.1.2 Storage Grain Controller

The storage grain controller runs as a user space daemon and implements storage virtualization; the details of managing storage grains are abstracted from the user. The controller therefore exports read, write and grain initialization interfaces to and receives access requests from the device driver. Internally, this process translates sector requests into communications with nearby storage grains. In order to support such functionality, the daemon is responsible not only for reliable communication, but also for managing resources usage, allocation strategies, and scheduling.

Because of the limited capabilities available to storage grains, as discussed in the next section, the complexity of underlying communication protocols must be kept to a minimum. The controller must therefore implement its own mechanisms to ensure correct read, write and initialization functionality. The details of these and their supporting protocols are defined in Section 3.3.

Managing the assignment and use of storage grains is accomplished through maintenance of a Sector Allocation Table (SAT). As shown in Figure 3, the SAT creates a mapping between the data sectors of the logical storage device and byte offsets in individual storage grains. Through a hash table lookup, incoming read and write requests are directed to the appropriate storage grain locations. Changes to the SAT itself occur only on the return from a successful write. The strategy by which data is actually mapped to storage grains is implementation dependent and should be based on workload. Section 4 provides insight by comparing the performance of a number of mapping polices.

The current prototype implementation only supports a FIFO scheduling algorithm; however, the benefits of traditional scheduling algorithms do not necessarily directly translate to this environment. Whereas spatial locality is often beneficial to request scheduling in traditional storage systems, competition for wireless resources by neighboring storage grains may in fact decrease system throughput. Spatial diversity, in combination with the ability to access nodes without mechanical repositioning, may in fact lead to better overall system performance. We leave a more in-depth investigation of this issue to future work.

3.1.3 Storage Grains

Memory spots are an emerging storage platform first announced in 2006. Characterized by their small form factor ($< 2mm^2$) and ability to communicate wirelessly, these devices are each capable of storing between 256KB and 4MB of data [8, 12, 18]. Their proposed uses range from embedding patient medical information in hospital wrist bands to storing supplementary multimedia material directly on paper documents. With access rates of approximately 10 Mbps, the ability to transparently carry and instantly access non-trivial quantities of data is very much a reality.

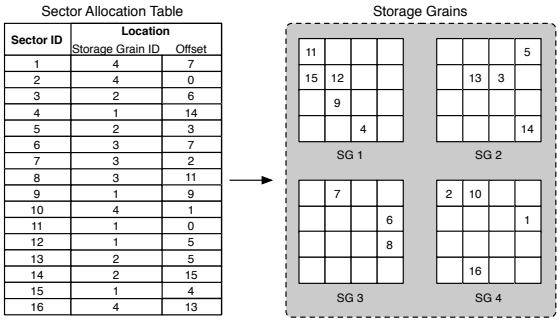

Figure 3: Example Sector Allocation Table and corresponding storage grains

Similar to Radio Frequency Identification (RFID) tag technology, memory spots are simple devices capable of executing a small amount of fixed functionality. Their lack of an independent power source limits interaction with such devices to a query/response relationship initiated by inductive coupling. Accordingly, each communication session with a memory spot is limited to a single message in each direction. While the documented range of communication by such devices is targeted to approximately 1mm, numerous examples of devices capable of extending such limits by orders of magnitude are well documented [37, 20]. Unlike RFID devices, however, the contents of memory spots can be rewritten. Accordingly, memory spots provide many of the attractive characteristics found in traditional storage media.

As the base components of a SAN, storage grains act as banks of storage sectors. These devices therefore perform a limited set of operations including the execution of read, write and initialization requests. Because space and power are scarce resources, restricting the functionality to support only a small set of instructions is necessary.

Memory spots are not available to the public as of the writing of this paper. Regardless, sufficient technical details have been made available such that their operations and functionality can be effectively emulated as independent processes in a conventional operating system. We base the behavior of our virtual storage grains, discussed in greater detail in Section 4, on this information.

3.2 Organization

One of the systemic challenges of a storage grain system is the management of sectors. Unlike traditional systems in which sector assignments are dictated by physical disk characteristics, the proposed system must man-

age the shifting geometry represented by the potentially transient population of storage grains. Our approach is to allocate sectors upon write, and re-allocate where recovery is needed (after grain failure or removal). This section considers solutions for allocation, leaving issues of recovery and reallocation to Section 3.4 and future work.

Storage grains are simple read/write storage devices. Each grain has a unique identifier burned into its memory by the manufacturer. All communication with a grain is tagged with this unique ID and the targeted device processes only messages containing its ID. The grains export a simple access interface: they receive and process reads/writes of arbitrary size to/from a specified offset within a contiguous address space. We abstract this storage interface to project an array of sectors[2] to accommodate the operating system storage interfaces. Storage operations are treated as single transactions involving an individual sector.

There are physical reasons that read operations are restricted to single sectors. As previously discussed, storage grains passively induce the energy used to transmit responses in a similar manner to RFIDs. Hence, each read request can only induce enough power for a storage grain to respond with a fixed amount of data. We conservatively chose to model power as limited to a single sector–in practice the grains made available in the future may induce enough energy for more than one sector. To be sure, environmental factors such as distance and host transmitter strength will vastly affect response capabilities, and its future study is essential to the tuning of the granular storage systems. To ease our initial development and analysis, we similarly restricted writes to single sector transactions.

Allocation is a conceptually simple process: from a population of storage grains of a given size[3], we select a particular storage grain with available space. The consequences of such an algorithm are profound. The performance and reliability of the system will be largely dictated by the allocation strategy. We implement the following sector allocation algorithms in our prototype:

- **linear** - This approach allocates storage from the first to last available sector on a given storage grain before writing to another device. In practice, all allocation requests use the lowest unused sector offset on the lowest available storage grain ID. This ensures that sectors will be concentrated on as few grains as possible.

- **striping** - This algorithm implements a round-robin approach in which the system allocates sectors evenly across available grains. To achieve a uniform distribution, new write requests are directed towards storage grains with the least number of used sectors.

- **random mapping** - Sectors are allocated to a random unused sector on a random storage grain. This ensures nothing other than, probabilistically speaking, uniform use of all sectors on all storage grains over a long term period.

These algorithms represent the three general philosophies of allocation–minimal spread, maximal spread, and random. There are opportunities to extend these simple algorithms to optimize different aspects of the system. For example, when a particular allocation strategy results in low throughput, sectors can be actively or opportunistically relocated to better respond to observed or expected workloads needs. We believe such performance characteristics to be core to understanding grain storage behavior, and its study is likely to uncover many interesting performance and engineering tradeoffs.

We consider the performance of the storage system built on each of the above algorithms under various workloads in Section 4.

3.3 Security

The architecture described above introduces a number of security challenges. Unlike traditional storage systems–which can depend on at least limited guarantees of physical security–the portable nature of storage grains makes them acutely vulnerable to physical compromise. Worse still, the use of wireless communications between storage grains and hosts allows a nearby adversary the ability to monitor, modify and insert messages into communications. We address these concerns by designing and implementing a series of protocols that provide secure access to storage grains. Note that because of similar underlying technology and constraints, we borrow liberally from the ideas (if not content) of RFID security protocols [32, 17, 15, 16, 27].

This section defines a simple security model for the granular storage system and outlines a system and protocol suite. We focus on the security of the interface between the host and storage grains, deferring issues of host-level security (e.g., file access control, user management) to the operating system upon which it rests. We use the following notation throughout:

- C is a controller.
- ID_{SG} is a device identifier.
- $K_{F,D}$ is a key specifying a function and device.
- K_G is a global data key.
- l is the length of a request.

[2]By default, the sectors are 512 bytes, but other sector sizes are supported by our prototype through compile-time configuration parameters.

[3]The current work assumes a uniform size for all storage grains. However, support for non-uniform storage grains could be added with trivial extensions to the allocation algorithms.

Initialization

1a. $\sigma = \{K_{W,SG}|K_{R,SG}\}_{K_{M,SG}}$ *(key confidentiality)*
1b. $h = \text{HMAC}_{K_{W,SG}}(ID_{SG}, n, \sigma)$ *(message integrity)*
1. *C* → *SG* : (*INIT_REQ, tn, l*) ID_{SG}, n, σ, h *(initialization request)*

2a. $h = \text{HMAC}_{K_{M,SG}}(succ, n)$ *(message integrity)*
2. *SG* → *C* : (*INIT_RESP, tn, l*) $succ, n, h$ *(counter value response)*

Discovery

3. *C* → * : (*HELLO_REQ, tn, l*) *(hello broadcast)*
4. *SG* → *C* : (*HELLO_RESP, tn, l*) $ID_{SG}, size$ *(hello response)*

Counter Sync

5. *C* → *SG* : (*COUNTER_REQ, tn, l*) ID_{SG} *(counter value request)*
6. *SG* → *C* : (*COUNTER_RESP, tn, l*) *counter_val* *(counter value response)*

Read

7a. *sdata* = $(\{data\}_{K_G}|\text{HMAC}_{K_G}(\{data\}_{K_G}))$ *(data confidentiality & integrity)*
7b. $h = \text{HMAC}_{K_{R,SG}}(ID_{SG}, n, offset)$ *(message integrity)*
7. *C* → *SG* : $(READ\_REQ, tn, l)$ $ID_{SG}, n, offset, h$ *(read request)*

8a. $h = \text{HMAC}_{K_{W,SG}}(succ, n, sdata)$ *(message integrity)*
8. *SG* → *C* : $(READ\_RESP, tn, l)$ $succ, n, sdata, h$ *(read response)*

Write

9a. $h = \text{HMAC}_{K_{W,SG}}(ID_{SG}, n, offset, sdata)$ *(message integrity)*
9. *C* → *SG* : $(WRITE\_REQ, tn, l)$ $ID_{SG}, n, offset$, $sdata, h$ *(write request)*

10a. $h = \text{HMAC}_{K_{W,SG}}(succ, n)$ *(message integrity)*
10. *SG* → *C* : (*WRITE_RESP, tn, l*) $succ, n, h$ *(write response)*

Figure 4: Storage grain secure communication protocols

- $n = (r, c)$ is a nonce, composed of a random number r and a counter c.
- SG is a storage grain.
- tn is a transaction number.

Note that due to the resource constraints, the use of public-key cryptography by grains is not possible. Hence, all cryptographic operations defined in this section are built upon symmetric-key cryptosystems, e.g., DES [28], AES [7]. Because such operations can be supported in RFID tags [6], we assume they are implemented in each of the storage grains.

Our storage system contains three classes of principals: *owners, readers* and *writers*. Owners are the proprietors of physical devices and, through the use of the master key $K_{M,SG}$, establish keys for reading ($K_{R,SG}$) and writing ($K_{W,SG}$). Ownership is regulated by possession of the master key. Master keys are assigned and burned into each storage grain by the manufacturer and delivered to the owner at time of purchase. All master keys associated with the storage grains owned by a host are manually configured on that host, e.g., placed in the local file. Non-owner hosts permitted to access a storage grain are given the appropriate read and/or write keys out of band.

All data written to grains by a host are confidentiality and integrity protected with a global data key K_G. This key is *not* known to the storage grains. These additional protections are used to preserve confidentiality and integrity in the presence of physical attacks on the storage grains, e.g., directly reading the flash memory, and defending against the loss of the master keys. This approach does not prevent *denial of service* attacks caused by channel saturation or physical erasure or corruption.

Ensuring that messages are not replayed (i.e., ensuring freshness) is accomplished using a two-component nonce $n = (r, c)$ passed with each transaction. The first part, r, is a random value ensuring message uniqueness. The second component, c, is a counter maintained by each storage

grain to validate nonce values. Each access to a storage grain must include a nonce value of at least c in order to be processed. The inclusion of r prevents an adversary from replaying stale data to a controller C with an incorrect c value. The protocol for synchronizing counter values between the controller and storage grain is described below.

Detailed in Figure 4, we now describe the security protocols used to communicate between the controller and storage grains. Storage grains are initialized via a series of maintenance operations. The owner of each device establishes read and write keys via the **initialization protocol**. In this, the owner transmits an initialization request to each of its storage grains containing encrypted copies of $K_{R,SG}$ and $K_{W,SG}$. Note that an owner can revoke read or write access by reinitializing keys at any point after the system starts. Controllers, which may or may not necessarily be an owner, receive the necessary read and write keys out of band. Controllers execute a **discovery protocol** to detect the presence of nearby storage grains. In this, C broadcasts a `HELLO` message and receives responses containing device ID (ID_{SG}) and size from its neighbors. Finally, C obtains the most recent counter value via the **counter sync** protocol.

The **read protocol** is used to extract written blocks from the storage grain. A read request contains the identifier of the destination grain, nonce, offset value and an HMAC. An access attempt is permitted if the requested block is valid (legal location and block had previously been written) and the HMAC validates. Upon receiving a valid message from the controller, a storage grain responds with a success flag, requested data and a hash of the transaction. If the controller receives a correct response, it decrypts the data using K_G and forwards it to the user.

The **write protocol** is used to place blocks on the storage grain. As described in Section 3.2, the block allocation strategy determines where the block should be placed. The controller then transmits a request containing the destination identifier, nonce, offset, data encrypted under K_G and an HMAC of the transaction. The receiving storage grain determines the validity of the response, stores the encrypted data and returns an HMACed transaction message. On successful writes, the client then updates the SAT to reflect changes in the contents of storage grains.

3.4 Reliability

Reliability is a key requirement of any storage system: the ability to accurately place and retrieve data under normal and even exceptional circumstances is of paramount importance. One seemingly attractive approach in granular storage systems is to simply use RAID [29]. In particular, features such as RAID 1 (mirroring) and RAID 5 (parity) can achieve fault tolerance across the devices by exploiting independent failures and removals of storage grains. Such previous schemes are instrumental in understanding and meeting these challenges, but as detailed below, the effectiveness of a direct application of the standard RAID approaches is unclear.

Granular storage systems must be exceptionally resilient to data loss. In fact, it is the intent of the system to be able to dynamically remove storage grains for use in other storage systems. For example, our protagonist in Section 2 transferred grains from the laptop storage system to enable his vehicle to use additional maps for his trip. Hence, more than just failure recovery is required; the system must survive potentially frequent failures and removals of subsets of the storage grains.

Disks fail in RAID as large associations of sectors. Because those associations are known *a priori* and because failures are rare, solutions such as parity are particularly effective. However, because storage grains are smaller, sectors fail in smaller association and in larger quantities. Thus, the kinds of failures one will encounter are likely to be more difficult to address.

Building on past storage reliability approaches, we propose to use a combination of mirroring and error correcting codes (of which parity bits are an example) to achieve a configurable level of reliability. We consider each of these techniques in turn below, and then consider the potential use in combination. Note that our efforts here are very preliminary: we only seek to characterize the problem and posit initial solutions.

Mirroring places multiple copies of the same data on separate devices. In the context of this work, the placement of a single sector on multiple devices increases the chances of the block being available after failure or removal of some of the storage grains. This leads to a simple calculus: the cost of implementing a storage system of size k with m mirrored copies of each block is $k * m$ (as measured in number of storage grains). Accordingly, the system cost grows linearly with the number of mirrored copies.

Error correcting codes [14] offer an alternative approach to reliability. For example, in the single or double parity approach used in RAID, one or two disks are used as "parity"–the loss of *any* other disk(s) can be repaired by consulting one or both parity disks. Such solutions can be extended to be resilient to potentially many failures, where each new parity disk represents an increase of resilience to one additional failure. Such systems increase the size of the system by a constant factor n, where the total system cost is $m + n$.

Now consider the effectiveness and runtime cost of each schemes. Assume the probability of any grain failing or being removed in a mirroring solution is r. Then, the probability of any given sector being unavailable is:

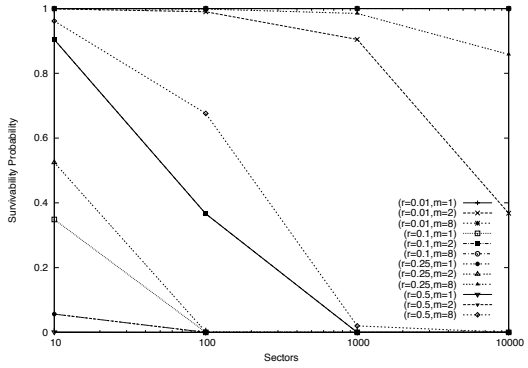

Figure 5: Storage survivability probability - r percent of storage grain failures in system with m sector mirrors.

$Pr(r)^m$. Such an approach may initially appear to be quite effective. For example, assume $Pr(r) = 0.25$ and $m = 6$, the probability that any given sector would be unavailable would be approximately 0.02%, or one fiftieth of a percent. However, we care about the reliability of the entire storage system. Thus, the real measure of the reliability would be the joint probability that all the sectors (where s is the number of unique sectors) were available, i.e., the survivability rate would be $(1-Pr(r)^m)^s$. A system with $s = 100$, $Pr(r) = 0.25$ and $m = 6$ would have over a 97% chance of being completely available. A system with the same total number of sectors but with half the mirroring (i.e. $s = 200$ and $m = 3$), however, would offer only a 4% chance of being reliable. As illustrated in Figure 5, these calculations indicate the presence of a "knee" after which the reliability of the solution drops from acceptable to non-existent as a function of the number of sectors.

Conversely, consider an n out of m error correcting code. The probability of availability for any sector being available is 100% if $m - n$ grains or less are lost, and 0% otherwise. However, the cost of resilience in this case is in additional writes—any update of a grain means the updating of all m EC codes. So, the cost of performing the writes is $m + 1$. Note that the parallelism afforded by storage grains may mitigate this cost.

The number of grains, their small sizes, and their expected failure rates mandate a careful inspection of these solutions. As we will explore in future work, understanding how we balance these two solutions and use them in combination both statically (at format time) and later (in response to observed failures/removals) to achieve target levels of reliability is central to making these systems and applications that will build upon them reliable.

4 Evaluation

The granular storage controller combines grains to export a logical storage system that is simply a linear collection of sectors. As previously discussed, translation between sectors and the physical location on a grain is defined by the SAT. Section 3.2 proposed linear, striping, and random allocation strategies and claimed the inherent parallelism of the latter two strategies would provide increased performance. We now explore this claim by implementing the granular storage system and evaluating the three allocation strategies under a variety of workloads.

Operating systems access block devices with sequences of sector requests corresponding to higher level workloads. In the absence of a scheduling algorithm, a sequential file read results in a bounded, in-order sequence of contiguous addresses. Out of order sector request sequences can also occur, typically as the result of simultaneous access to many files. Such sequences may also be the result of higher-level storage system inefficiencies such as file system fragmentation. That is, a sequential file read does not always result in an in-order sector request sequence.

Given linear, striping, and random allocation strategies, the granular storage system will react differently to specific workloads. A storage grain can only process one request at a time; if the next sector request is assigned to a storage grain currently processing a request, the controller will stall until that grain is free. Because the linear allocation strategy assigns adjacent sectors to the same storage grain, a sequential file access should experience severe performance degradation. The stripe and random allocation strategies therefore were designed to increase throughput by distributing adjacent sectors amongst available storage grains. Out of order sector request sequences result in non-repeating grain requests regardless of the allocation strategy, thereby potentially improving the linear allocation strategy. At the same time, out of order sector request sequences will degrade the stripe allocation strategy performance due to unmet workload assumptions. Simply put, the granular storage system throughput is directly proportional to the variety of storage grains accessed.

4.1 Experimental Setup

Our implementation of the granular storage system provides a mechanism to evaluate file system request workloads. The kernel block driver is functional but not optimized for performance[4]. Therefore, our experiments focus directly on the controller. Sector requests are produced via a specialized load generator and sent directly

[4]The camera ready version of this paper will include performance results using the block driver.

to the controller. We use the load generator to produce sequences for six experiments covering the range of system workloads discussed above.

The **Small-File In-Order** workload produces a short, in-order sequence beginning at a random location within the entire storage space. This sequence corresponds to the operating system performing a linear read of a small file. The experiment is repeated using 1, 2, 4, 8, 16, 32, and 64 grains. Test runs use a sequence equivalent to a 100KB file. This size represents a small system file and is small enough to be stored on a single grain under the linear allocation strategy.

The **Small-File Random-Order** workload is similar to the in-order variant. Instead of producing an in-order sequence, the test uses a random permutation. The permutation corresponds to an out of order sequence resulting from system peculiarities such as fragmentation. The experiment is repeated over the same number of grains as the small-file in-order workload and also uses a sequence equivalent to a 100KB file.

The **Large-File In-Order** workload considers sequences resulting from linear requests of large files. It has the same motivations as the small-file variant, except it uses a 10MB file instead of 100KB. The experiment is only performed using 16, 32, and 64 grains due to file system size restraints; each grain is modeled as providing only 1MB of storage.

The **Large-File Random-Order** workload mirrors the in-order variant, but includes the sequence shuffling described for the small-file random-order workload. This workload mirrors the operation of a database system. The randomized sequence order spans many grains, therefore the effects of parallelism will be observed even for the linear allocation strategy. Just as with the large-file in-order workload, the experiment is only performed with 16, 32, and 64 grains.

The **Many-File In-Order** workload considers multiple file access. Individual file accesses produce in-order sequences, each beginning at a random sector within the entire storage space. The accesses are interleaved to produce a widely varying request sequence. The experiment load represents twenty 100KB files and performs the experiment only over 4, 8, 16, 32, and 64 grains do to file system size restraints.

The **Many-File Random-Order** workload adds randomized permutations to in-order variant. The generated sequence provides the most random request order of the workload set. As the request order itself is essentially random, little difference between the allocation strategies should be observed.

Each workload was performed for the three allocation strategies and a varying number of storage grains. Our testbed included a set of five Linux systems running a recent version of the 2.6 kernel. Four systems ran the storage grain emulators and one acted as the controller. Experimental runs requiring more than four grains distributed the remaining grains amongst the four systems running the grain emulators. For the entire test duration, each grain emulation host ran 16 instances of the software, however, not all instances were used in each experiment run. The grain emulation hosts varied in performance potential. Two grain emulation hosts had 1.73GHz Intel Pentium M processors and 1GB RAM. A third contained a 3.0Hz Intel P4 processor and 2GB RAM. The fourth grain emulation host used a 2.8GHz Intel P4 processor and 1GB of RAM. The controller ran on a system with a 3.2GHz Intel Xeon processor and 2GB of RAM.

The grain emulation hosts and controller were connected via an isolated 10/100Mb/s Fast Ethernet switch. Experiments were performed on a wired network in order to provide a controlled setting impervious to environmental factors. Our evaluation goals are to observe the effects of various allocation strategies and not contention over a wireless medium, therefore the wired setting is appropriate. We do however, contrast the latency resulting from wired and wireless transmission media in Section 4.3 using the Bluetooth variant of our implementation.

4.2 Results

The experimental results provided in Figures 6 through 11 are the average of 100 runs of the workload experiments for each allocation strategy. All experiments had 95% confidence intervals less than two orders of magnitude smaller than the mean. Each run was randomly a set of read or write requests. Timers were inserted into the code to measure the duration of the run. The total request size was then divided by the duration to produce the throughput presented in the figures.

Figure 6 shows the experimental throughput for the small-file in-order workload. As expected, the linear allocation strategy performs significantly worse than the stripe and random strategies. The allocation strategy translates a sequence of sector requests into a sequence of grain requests. The linear strategy translates the in-order workload into a series of sequential requests to the same grain. As grains can only process one request at a time, the controller must wait for a response between each sector request. Note that if storage grains are capable of processing multi-sector requests, the linear allocation strategy could be more viable. However, our analysis assumes a grain can only process one sector request at a time. Hence, Figure 6 clearly demonstrates the throughput potential of allocation strategies that distribute requests amongst available storage grains. Throughput is more than doubled in some cases.

The small-file workloads shown in Figures 6 and 7 show a throughput increase as the number of storage

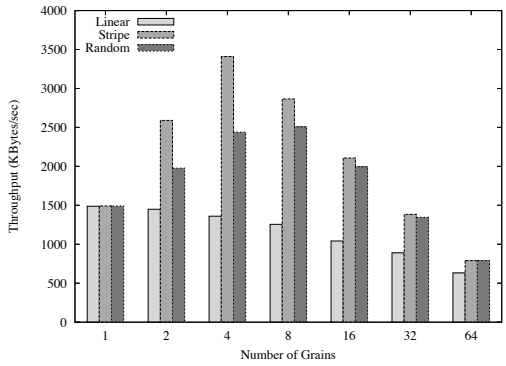

Figure 6: Small-File In-Order Workload Throughput (100KB file)

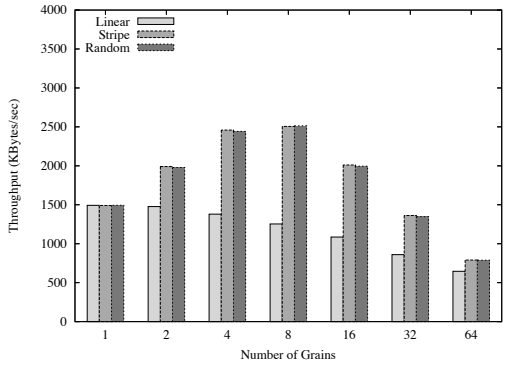

Figure 7: Small-File Random-Order Workload Throughput (100KB file)

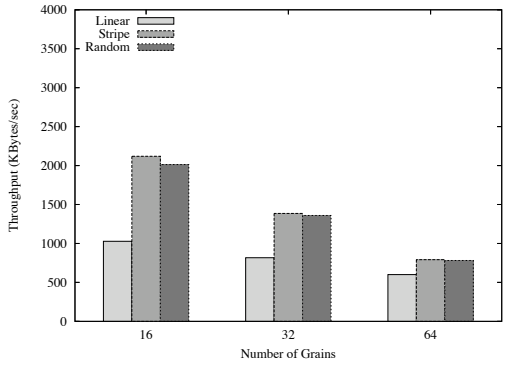

Figure 8: Large-File In-Order Workload Throughput (10MB file)

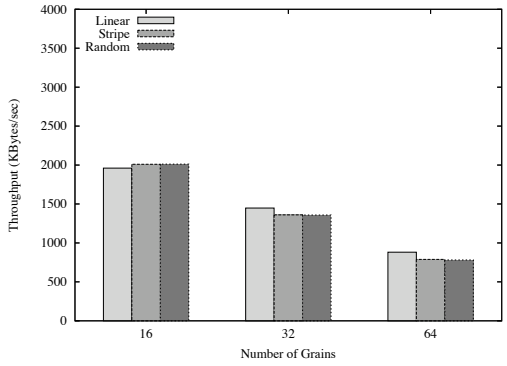

Figure 9: Large-File Random-Order Workload Throughput (10MB file)

grains increase; however, after a point, the throughput begins to decrease. We believe the bell curve is an artifact of the testing apparatus. Only four hosts emulate storage grain instances. Resource contention, e.g., processor contention and context switching, degrades performance when one host processes requests for more than one storage grain. The bell curve for the stripe strategy in Figure 6 confirms this suspicion. The stripe allocation strategy perfectly distributes sector requests to available grains. The curve peaks at four storage grains. Coincidentally, this is the same as the number of grain emulation hosts. Therefore, we posit that increasing the number of grain emulation hosts will prolong the bell curve peak. Note that the peak is slightly translated to the right for the random allocation strategy in Figure 6 and both stripe and random strategies in Figure 7. This phenomenon likely results from randomness in the allocation strategy and/or the workload, causing an increase in the apparent parallelism of the requests and consequently relieving contention on grain emulation hosts.

Randomness performs a significant role in observed throughput. In Figure 6, the stripe allocation strategy is easily distinguishable from the random strategy. However, ostensibly no difference exists in Figures 7 through 11. Table 1 quantitatively explores the differences between workloads for each allocation strategy. The table uses the small-file in-order workload as a baseline for comparison, as it provides optimal performance for the stripe allocation strategy. The percent difference in throughput is shown for the other workloads. Looking specifically at the stripe strategy, all workloads incorporating randomness result in a throughput decrease (recall that the many-file in-order workload starts each file at a random location and interleaves sector requests). Logically, without a perfect input sequence the stripe strategy does not evenly distribute requests to grains. Real system loads rarely conform to perfect in-order sequences, therefore the stripe and random allocation strategies are expected to perform similarly under typical workloads.

Workload randomness hurts the stripe strategy perfor-

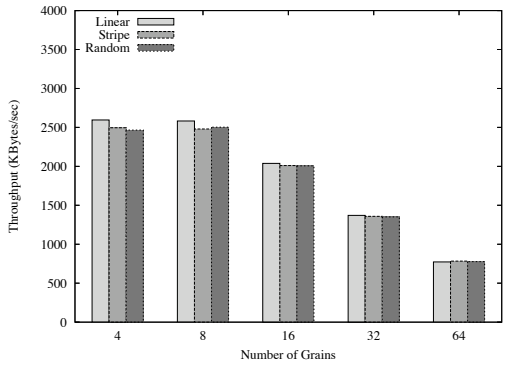

Figure 10: Many-File In-Order Workload Throughput (20, 100KB files)

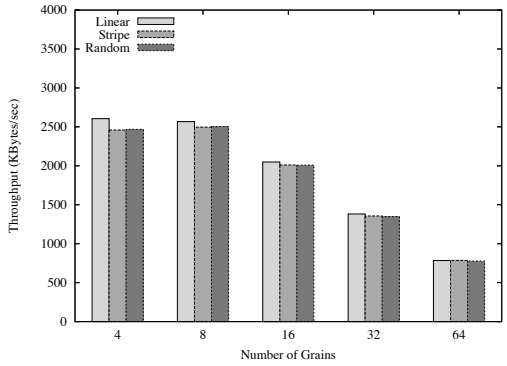

Figure 11: Many-File Random-Order Workload Throughput (20, 100KB files)

mance; however it aids the linear allocation strategy. Table 1 shows significant throughput increases, over 100% in some cases, for the linear allocation strategy in the face of randomized workloads. These workloads result in distributed grain requests, equivalent to those in the random allocation strategy. Observe that in Figures 9, 10, and 11, the linear strategy performs the same, if not marginally better than the other two strategies. Finally, the large-file workloads, Figures 8 and 9, strongly distinguish the effects of input randomization. No parallelism exists from the in-order workloads, hence performance remains low.

All allocation strategies perform equivalently for randomized workloads; however, the stripe allocation strategy throughput is increased for in-order sequences. Therefore, it provides a starting point for system designers wishing to implement a scheduling algorithm. However, the complexities of dynamic SAN membership from lost, broken and re-purposed storage grains will certainly impact the optimality of such a strategy. Moreover, optimizing a scheduling algorithm for the stripe allocation strategy requires more care than simply applying traditional mechanical disk schedulers. Optimal sector sequences must be more than in-order. The sequences producing the high throughput in Figure 6 were contiguous in addition to in-order. Therefore, to reach optimal performance, the scheduler must be cognizant of the number of storage grains associated with the file system and reorder requests accordingly. Furthermore, because performance of real storage grains will be affected by contention for the wireless medium, an optimized scheduler should understand storage grain geography.

Our granular storage system implementation allowed for an exploration of various sector allocation strategies. The stripe allocation strategy performs well for optimal sector request sequences, however, it converges on the random allocation strategy under realistic workloads. As shown in the results, randomized workloads cause the three allocation strategies to converge; even the linear strategy performs well due to the parallelism present in the workload. During the course of our experiments we found performance to decrease after the number of grains became greater than a certain value. We suspect this resulted from resource contention, as each grain emulation host ran multiple grain instances. We expect this trend will disappear as the number of emulation hosts increase. Finally, our evaluation only considered a wired interconnection network, as we do not know specifics of storage grain communication. In reality, storage grains will communicate wirelessly. Wireless communication is traditionally slower than wired communication, therefore we further explore request/response delay.

4.3 Access Latency

The experiments described in Section 4.1 evaluated the system impact of different allocation strategies; however, it did not provide insight into the communication overhead. One final experiment was performed to analyze the access latency between wired and wireless media. Bluetooth was chosen for wireless communications, as it provides a conservative estimation of lower bounds. We explored access latency by timing over 100,000 read and write requests from the controller to one remote grain. We first access the storage grain using the same wired network described in Section 4.1. The requests were then repeated between the same hosts, but over a Bluetooth wireless network connection. The Bluetooth socket API is slightly different than traditional network sockets, therefore our implementation required adjustments. Of particular note, the Bluetooth socket API required each endpoint to establish a connection before even "connectionless" messages can be sent. Finally, the experiment used only one storage grain to avoid packet loss due to packet collision.

Table 1: Comparison of Throughput between Workloads

Linear Allocation Strategy							
Workload	1 Spot	2 Spots	4 Spots	8 Spots	16 Spots	32 Spots	64 Spots
Small In-Order	1,488 KB/s	1,449 KB/s	1,361 KB/s	1,256 KB/s	1,042 KB/s	890 KB/s	632 KB/s
Small Random	+0.3%	+1.9%	+1.5%	-0.2%	+4.2%	-3.3%	+2.2%
Large In-Order	–	–	–	–	-1.3%	-8.2%	-5.1%
Large Random	–	–	–	–	+88.2%	+62.8%	+39.4%
Many In-Order	–	–	+90.6%	+105.6%	+95.7%	+54.0%	+22.4%
Many Random	–	–	+91.5%	+104.4%	+96.7%	+55.3%	+24.1%
Stripe Allocation Strategy							
Workload	1 Spot	2 Spots	4 Spots	8 Spots	16 Spots	32 Spots	64 Spots
Small In-Order	1,492 KB/s	2,590 KB/s	3,410 KB/s	2,866 KB/s	2,107 KB/s	1,384 KB/s	790 KB/s
Small Random	-0.1%	-23.1%	-27.9%	-12.6%	-4.6%	-1.6%	+0.2%
Large In-Order	–	–	–	–	+0.6%	+0.1%	+0.3%
Large Random	–	–	–	–	-4.6%	-1.7%	-0.1%
Many In-Order	–	–	-26.8%	-13.5%	-4.6%	-1.8%	-0.7%
Many Random	–	–	-27.8%	-12.8%	-4.6%	-2.0%	-0.4%
Random Allocation Strategy							
Workload	1 Spot	2 Spots	4 Spots	8 Spots	16 Spots	32 Spots	64 Spots
Small In-Order	1,492 KB/s	1,977 KB/s	2,440 KB/s	2,508 KB/s	1,994 KB/s	1346 KB/s	790 KB/s
Small Random	+0.1%	+0.1%	+0.1%	+0.3%	+0.1%	+0.1%	-0.2%
Large In-Order	–	–	–	–	+1.0%	+1.0%	-1.1%
Large Random	–	–	–	–	+0.9%	+0.9%	-1.1%
Many In-Order	–	–	+1.1%	-0.2%	+0.6%	+0.3%	-1.7%
Many Random	–	–	+6.8%	+2.4%	+2.8%	+2.6%	-0.8%

Table 2: Measured Access Latency

Medium	Read	Write
Wired	$327.2 \pm 0.2\ \mu s$	$324.5 \pm 0.2\ \mu s$
Bluetooth	$22.44 \pm 0.03\ ms$	$20.98 \pm 0.03\ ms$

The experiment results are summarized in Table 2. In both the wired and Bluetooth cases, read and write requests exhibited nearly identical delays. Finally, as expected, Bluetooth was much slower than the wired network. The experiment showed Bluetooth to be approximately two orders of magnitude slower. We expect these values to offer approximate upper and lower bounds for future storage grain devices.

5 Related Work

Distributed networked file systems (e.g., NFS [35], AFS [13], and CIFS [23]) are organizations of file systems over a number of networked data servers in which data can be shared by a number of users [24]. The reliability and performance of these systems have been examined in systems like CODA and VESTA. Through the use of local user caching and replication of data, the CODA system [21] is able to support disconnected operation. The VESTA system [5] partitions data into disjoint sequences in order to parallelize accesses to data.

In contrast to distributed networked file systems, storage systems such as storage area networks (SANs) [30], manage data at the block level. SANs can be configured as RAIDs [29], which distribute the organization of data across storage nodes in order to improve performance and reliability [9]. Such storage systems are commonly built over iSCSI [33] or Fibre Channel [30] network communication infrastructures.

General storage security has been analyzed with regards to the way in which data is protected on both the communication and storage media in various systems [31]. Single server storage security has largely been implemented such that data is encrypted on the storage device itself [2, 26]. Conversely, distributed file systems provide both access control and protect data while in transmission, but do not encrypted stored data. For example, AFS uses Kerberos for authentication and secure RPC for the protection of communicated data [34]. Similarly, NASD, network attached secure disk, and secure NFS provide mechanisms accomplishing the same ends [10, 11, 36]. In addition to achieving these objectives, the Secure File System (SFS) presents a novel key

distribution mechanism [25]. Techniques addressing similar security concerns in distributed storage systems have also been suggested for iSCSI [19, 4, 22].

Similar to distributed file and storage systems, significant research has also been done on the security of RFID technology. RFID security goals include not only protecting tag data [16, 17], but ensuring owner privacy by hindering the ability for an adversary to track, clone or impersonate the tag [1, 15, 27, 32]. Such tasks are difficult due to the limited ability to perform complex functions including cryptography [3].

6 Conclusion

Traditional storage systems are designed to support particular sets of operations. Devices capable of high performance and capacity generally lack portability. By contrast, mobile devices are often incapable of benefiting from the advantages of parallelism. Emerging memory spot technology, however, offers the potential to overcome these seemingly inherent inflexibilities. By constructing a storage area network (SAN) from an array of miniature wireless devices, each emulating the functionality of individual disk tracks, the creation of malleable storage devices becomes possible. Our granular storage system recognizes and addresses core requirements including organization, security and reliability. Through a thorough examination of system behavior and performance, we characterize the impact of such design requirements and how traditional scheduling and allocations strategies fail to map to this medium. As a result, we demonstrate the ability to construct an arbitrarily reconfigurable logical storage system built on storage grains. While significant design challenges exist, this work creates a base architecture upon which a diverse set of new applications can be constructed.

References

[1] G. Avoine and P. Oechslin. RFID Traceability: A Multilayer Problem. In *Financial Cryptography and Data Security*, 2005.

[2] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the ACM conference on Computer and Communications security*, 1993.

[3] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, and M. Szydlo. Security Analysis of a Cryptographically-Enabled RFID Device. In *Proceedings of USENIX Security Symposium*, 2005.

[4] S. Chaitanya, K. Butler, A. Sivasubramaniam, P. McDaniel, and M. Vilayannur. Design, implementation and evaluation of security in iSCSI-based network storage systems. In *Proceedings of the second ACM workshop on Storage security and survivability*, pages 17–28, 2006.

[5] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[6] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong Authentication for RFID Systems using the AES Algorithm. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2004.

[7] FIPS. Advanced encryption standard (AES). Federal Information Processing Standards Publication 197, 2001.

[8] I. Genuth. HP's Memory Spot Chip is Spot On. `http://ww.tfot.info/content/view/79/59/`, 2006.

[9] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[10] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, 1999.

[11] H. Gobioff, D. Nagle, and G. Gibson. Embedded Security for Network-Attached Storage. Technical Report CMU-CS-99-154, Carnegie Mellon University, 1999.

[12] Hewlett-Packard. HP Unveils Revolutionary Wireless Chip that Links the Digital and Physical Worlds. `http://www.hp.com/hpinfo/newsroom/press/2006/060717a.html`, 2006.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions Computer Systems*, 6(1):51–81, 1988.

[14] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *SIAM Journal of Applied Mathematics*, 8:300–304, 1960.

[15] A. Juels. Strengthening EPC tags against cloning. In *Proceedings of the 4th ACM workshop on Wireless security*, pages 67–76, 2005.

[16] A. Juels. RFID security and privacy: a research survey. *IEEE Journal on Selected Areas in Communications (JSAC)*, 24(2):381–394, 2006.

[17] A. Juels, R. L. Rivest, and M. Szydlo. The blocker tag: selective blocking of RFID tags for consumer privacy. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2003.

[18] M. Kanellos. HP's Memory Spots put video, audio into photos. `http://news.com.com/HPs+Memory+Spot+puts+video,+audio+into+photos/2100-11392\_3-6094586.html`, 2006.

[19] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC2401, 1998.

[20] I. Kirschenbaum and A. Wool. How to Build a Low-Cost, Extended-Range RFID Skimmer. In *Proceedings of the USENIX Security Symposium*, 2006.

[21] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the CODA file system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1991.

[22] Y. Klein and E. Felstaine. Internet Draft of iSCSI Security Protocol. `http://quimby.gnus.org/internet-drafts/draft-klein-iscsi-security-00.txt`, 2000.

[23] P. Leach and D. Perry. CIFS: A Common Internet File System. *Microsoft Internet Developer*, 1996.

[24] E. Levy and A. Silberschatz. Distributed file systems: concepts and examples. *ACM Computing Surveys*, 22(4):321–374, 1990.

[25] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 124–139, 1999.

[26] E. Miller, D. Long, W. Freeman, and B. Reed. Strong Security for Network-Attached Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[27] D. Molnar, A. Soppera, and D. Wagner. A Scalable, Delegatable Pseudonym Protocol Enabling Ownership Transfer of RFID Tags. In *Selected Areas in Cryptography*, 2005.

[28] National Bureau of Standards. Data Encryption Standard. *Federal Information Processing Standards Publication*, 1977.

[29] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM International Conference on Management of Data*, 1988.

[30] B. Phillips. Have storage area networks come of age? *Computer*, 31(7):10–12, 1998.

[31] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage Security. In *Proceedings of the USENIX Conference on File and Storage Security (FAST)*, 2002.

[32] S. E. Sarma, S. A. Weis, and D. W. Engels. RFID Systems and Security and Privacy Implications. In *Cryptographic Hardware and Embedded Systems*, 2002.

[33] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC3720.

[34] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[35] S. Shepler, B. Callaghan, R. Turlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC3530.

[36] B. Taylor and D. Goldberg. Secure Networking in the Sun Environment. In *USENIX Summer*, pages 28–37, 1986.

[37] Tom's Hardware. How To: Building a BlueSniper Rifle. `http://www.tomsnetworking.com/Sections-article106.php`, 2005.