Secure, Energy-Efficient, Evolvable, Long-Term Archival Storage

Technical Report UCSC-SSRC-09-01
March 2009

Mark W. Storer
mstorer@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SECURE, ENERGY-EFFICIENT, EVOLVABLE, LONG-TERM ARCHIVAL STORAGE

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Mark W. Storer

March 2009

The Dissertation of Mark W. Storer
is approved:

Ethan L. Miller, Chair

Mary G. Baker

Darrell D.E. Long

Kaladhar Voruganti

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Secure, Energy-Efficient, Evolvable, Long-Term Archival Storage

by

Mark W. Storer

Users are storing ever-increasing amounts of information digitally, driven by many factors including government regulations and the public's desire to digitally record their personal histories. Unfortunately, we have yet to demonstrate that we can reliably preserve digital data for more than a few years, putting a generation's cultural legacy at risk. Much of the problem is rooted in our approach to building long-term storage systems; currently archival systems are developed using the same approaches, access patterns and techniques used to design higher-performance, shorter-term storage systems. As a result, current archival storage systems still rely on strategies that fail in long-term scenarios, waste money and energy, and perpetuate the endless cycles of "fork-lift" upgrades and wholesale migrations needed to remain efficient and up to date.

In my thesis, I demonstrate that archival storage is a first class category of storage that requires specialized solutions. To this end, I present several techniques tailored specifically for the unique demands of long-lived data. To explore the security needs of archival data, I have developed POTSHARDS, which offers secrecy through unconditionally secure secrecy techniques, and survivability through increased attack detection and built-in data recovery. To study cost savings, I have created Pergamum, a distributed system of intelligent storage appliances that stores data reliably with multi-level encoding and a hierarchical auditing scheme, and energy-efficiently by leveraging existing MAID techniques, while extending them by exploiting the different access patterns of data and metadata. Running atop of Pergamum is Logan, a management layer being developed that actively identifies and decommissions wasteful devices in order to continuously maximize system efficiency. These systems combine to demonstrate significant progress towards effective, secure, energy-efficient, and evolvable archival storage.

To my mom, dad, and sister,

Acknowledgments

I would like to thank all the members of the Storage Systems Research Center (SSRC) who provided valuable technical feedback and assistance on developing the feedback in this thesis. More importantly, on a personal level, the moral support they have offered over the years has been invaluable. Hopefully the constant complaining made by myself and the senior students did not dissuade the newer students from following a career in systems research.

This research was supported by the Petascale Data Storage Institute under Department of Energy award DE-FC02-06ER25768, and by the industrial sponsors of the SSRC, including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Data Domain, Digisense, Hewlett-Packard Laboratories, IBM Research, LSI Logic, NetApp, Seagate, Symantec, and Yahoo!.

Chapter 1

Introduction

> Whereof what's past is prologue, what to come in yours and my discharge.
>
> William Shakespeare

The ability to store and maintain massive quantities of data is becoming increasingly important, as scientists, businesses, and consumers are increasingly aware of the value of archival data. Scientists have long attempted to preserve data archivally, though such efforts have sometimes fallen short. For businesses, data retention is mandated by law [2, 3], and data mining has proven to be a boon in shaping business strategy. For individual consumers, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies and personal documents [106, 107]. Unfortunately, traditional storage systems are not designed to meet the needs of long-term, archival data [18, 19, 119].

Paradoxically, despite the increasing value of archival data, high cost is one of the biggest obstacles to applying traditional storage techniques to design systems to house archival data; the goal of cost-efficient, long-term storage is to enable the potentially indefinite retention of all data that *might* one day prove useful [35, 103]. With current systems, it is simply too expensive to store everything indefinitely (if they can provide any long-term persistence guarantees at all). Archival storage therefore needs to inexpensive to obtain (static costs), inexpensive to operate (operational costs), easy to scale over time (evolvable), and secure enough to safely store private information indefinitely.

At the core of my thesis is the premise that archival storage is a distinct class of stor-

age that is poorly served by general-purpose storage systems [18]. More specifically, compared to traditional workloads, the long data lifetimes of archival storage marks a fundamental distinction that requires solutions specifically tailored to data with a potentially indefinite lifetime. I demonstrate this need by examining three areas of archival storage: security, cost efficiency and management. While solutions to all three problems exist within the scope of traditional, enterprise storage, my thesis demonstrates that, within an archival setting, all three require specialized solutions.

It deserves to be noted, however, that long-term digital storage is not a wholly technical problem, nor can it be answered with a wholly technical solution. Instead, digital preservation requires an examination of storage at a higher level than media, file systems or even storage systems; long-term preservation of digital information must involve critical thought at the level of people and organizations [34, 106, 107]. Truly, just as archival storage is well served by an evolvable solution that adapts gracefully over time, the human talents of digital custodians must also adapt. To that end, an important goal of evolvable archives is to preserve data long enough so that future advances in application preservation can be applied.

Long-term preservation of data is still a relatively young, and increasingly active area of study. The work presented in this thesis covers an on-going exploration of topics within archival storage, a fact reflected in the diverse range of maturity in the projects that comprise this study. POTSHARDS was constructed to investigate security and survivability for data with an indefinite lifetime. It is relatively mature work that has produced several publications. Pergamum was created to explore a reliable, cost-efficient archival storage architecture that aggressively realizes static and operational cost savings. Pergamum progressed fairly rapidly, and while mature enough to produce a fully formed publication, it is still relatively young. Logan is the youngest of the three projects, and was designed to explore archival storage management. It has only recently reached the stage where preliminary designs can be presented. Logan came about as a result of the capabilities enabled by Pergamum.

1.1 Security

While storage security has long been an active, well-researched area, the indefinite lifetimes of archival storage introduce a number of new challenges [19, 62, 162]. One of the biggest challenges is that mechanisms such as cryptography work well in the short-term, but

are less effective in the long-term. The use of computation-bound encryption in an archival scenario introduces the problems of lost keys, compromised keys, and even compromised cryptosystems. All this is exacerbated by the numerous key rotations and cryptosystem migrations that will inevitably occur over the course of several decades; this must all be done without user intervention because the user who stored the data may be unavailable. Thus, security for archival storage must be designed explicitly for the unique demands of long-term storage.

Since security covers a large set of properties, I focused on the long-term implications of a few fundamental features: data secrecy and data accessibility. More specifically, planning for long-term data storage requires an examination far above the level of the storage system itself, involving critical assessment at the organization level and higher. Thus, I focused my examinations on those aspects of secrecy and availability that traditionally make organizational assumptions that are invalid in long-term scenarios. First, for data secrecy, both internal and external attackers must be considered; even an altruistic organization can change over time to become a primary threat to data secrecy. Second, with indefinite data lifetimes, it is unreasonable to assume that key authorities will survive past the immediate future. For data accessibility this means that data must be available and accessible in plaintext form for valid users without relying on the survival of a third party for key management; it is just as unacceptable to recover only ciphertext, as it would be to reveal plaintext to unauthorized users.

To address the many security requirements for long-term archival storage, I have designed and implemented POTSHARDS (Protection Over Time, Securely Harboring And Reliably Distributing Stuff), which uses three primary techniques to provide security for long-term storage. First, secret splitting [**?**] is used to produce a tuple of *n secret shares* from a block of data, *m* of which must be obtained to reconstruct the block. Unlike encryption, secret splitting is unconditionally secure; it can be shown that combining fewer than *m* shares reveals *no* information about the original block. Second, POTSHARDS uses a global data namespace that is used to identify data entities. The namespace introduces an element of diversity [53] into the model, since it is sparsely populated and treated in a similar fashion to a heap. Third, POTSHARDS utilizes approximate pointers, which differ from traditional pointers in that they indicate a *region* in the namespace as opposed to an exact address. Approximate pointers enables secure recovery from only the data itself by associating related secret shares in a way that does not unduly compromise security.

1.2 Cost Efficiency

Many storage systems designed for long-term data preservation rely on sequential-access technologies, such as tapes, that decouple media from its access hardware. While effective for back-up workloads (write-once, read-rarely, newer writes supersede old), such systems are poorly suited to archival workloads (write-once, read-maybe, new writes unrelated to old writes). With as many as 50–100 tapes per drive, a requirement to keep tapes running at full speed, and a linear media-access model, random-access performance with tape-media is relatively poor. This conspires against many archival storage operations — such as auditing, searching, consistency checking and inter-media reliability operations — that rely on relatively fast random-access performance. This is especially important in light of the preservation and retrieval demands of recent legislation [2, 3]. Further, many data retention policies include the notion of a limited lifetime, after which data is securely deleted; selective deletion is difficult and inefficient in linear media. Finally, the separation of media and access hardware introduces the need to preserve complex chains of hardware; reading an old tape requires a compatible reader, controller and software.

Recently, hard drives have dropped in price relative to tape, making them a potential alternative for archival storage [126]. The availability of high-performance, low-power CPUs [15] and inexpensive, high-speed networks have made it possible to produce a self-contained, network-attached storage device [60, 138] with reasonable performance and low power utilization: as little as 500 mW when both the CPU and disk are idle. The use of disks instead of tapes means that heads are packaged with media, removing the need for robotics and reducing physical movement and system complexity. Using standardized communication interfaces, such as TCP/IP over Ethernet, also helps simplify technology migration and long-term maintenance. By using randomly-accessible disks instead of linear tapes, systems can take advantage of inter-media redundancy schemes. Unfortunately, many existing disk-based systems incur high costs associated with power, cooling and administration because of design approaches that favor performance over energy efficiency. However, recent work on MAIDs (Massive Arrays of Idle Disks) has demonstrated that considerable energy-based cost savings can be realized while maintaining high levels of performance [39, 122, 188], though such systems still often favor performance over even greater energy savings.

While my design leverages MAID techniques, it also extends it by removing the need

for centralized controllers, and by exploiting the different access patterns of data and metadata. Pergamum takes an approach similar to that used in high-performance scalable storage systems [144, 189, 192], and is built from thousands of intelligent storage appliances connected by high-speed networks that cooperatively provide reliable, efficient, long-term storage. Each appliance, called a Pergamum *tome*, is composed of four hardware components: a commodity hard drive for persistent, large-capacity storage; on-board flash memory for persistent, low-latency, metadata storage; a low-power CPU; and a network port. Each appliance runs its own copy of the Pergamum software, allowing it to manage its own consistency checking, disk scrubbing and redundancy group responsibilities. Additionally, the CPU and extensible software layer enables disk-level processing, such as compression and virus checking. Finally, the use of standardized networking interfaces and protocols greatly reduces the problem of maintaining complex chains of dependent hardware.

1.3 Management

In contrast to traditional storage systems, which are typically more concerned with scalability in performance and capacity, an archival system designed for long-lived data must scale over many dimensions, including time, vendors and technologies [19]. The goal, therefore, is to move away from an endless series of migrations and "fork-lift" upgrades, to a continuously evolving system. To realize this goal, I have begun development on Logan, a software-based management layer that runs atop a distributed architecture such as Pergamum [168]. While devices in such an architecture can operate independently, their full potential is realized when they cooperate in inter-device redundancy groups to provide data reliability and ensure data longevity. Further, since the storage nodes are intelligent, each device contains specialized software that acts as an abstraction layer between the system, and the device's underlying hardware. This flexibility provides the potential for an adaptable system that changes gradually with technology; while individual components may change, the overall system evolves gracefully.

Although a fully distributed system is well suited to an evolvable design, it introduces the problem of managing the global state of a fully decentralized system. It is impractical for each node to maintain global knowledge—keeping just 10 KB per node for each of a million nodes would require 10 GB of storage per node. Moreover, keeping that information current would require far too many messages to be exchanged between nodes. While recent work has

made great strides towards efficiently aggregating data over very large networks of distributed nodes [199, 200], even these approaches may be insufficient in a system that could easily encompass millions of nodes. Instead, a million-node distributed system must facilitate nodes joining the system, manage placement of data and redundancy information, handle node failure, and gracefully phase out nodes as they age with only partial knowledge of the whole system and more complete knowledge of a small part.

Further, as archival systems become more useful as their cost decreases, long-term storage management must recognize that energy efficiency requires constant, proactive optimizations. While some earlier systems have addressed energy efficiency [39, 68, 121, 122, 188], none have examined how opportunity costs affect a system over its lifetime. Since drive capacity, real estate values and power costs are always increasing, system efficiency must be measured against what is currently achievable, not simply what was *once* achievable. For example, most storage systems assume that drives are replaced due to failure or wholesale system upgrades, suggesting that drives may remain in use well past the point of being economically efficient. Further, proactive decommissioning could also improve system reliability; earlier work has shown that the previously-held bathtub failure model for hard drives may not be valid [147], and that even small numbers of sector failures can presage overall drive failure [16, 17].

The remainder of my thesis proceeds as follows. Chapters 2 and 3 provide a brief background on the subjects covered in this thesis, and place my work in the context of existing research. Following that, Chapter 4 presents POTSHARDS, a system I designed to explore long-term data secrecy and recoverability. Chapter 5 discusses Pergamum, a system I designed that provides reliable, cost-efficient archival storage in an architecture that enables system evolvability. Chapter 6 provides an overview of Logan, an early stage project that indicates current progress towards evolvable system management. Finally, in Chapter 7 I summarize my results, and conclude my thesis.

Chapter 2

Background

> Knowledge is of two kinds: we know a subject ourselves, or we know where we can find information upon it.
>
> Samuel Johnson

The purpose of this section is to present the background information needed to evaluate my work on archival storage. This involves a number of important topics. First, I present a discussion on storage security, and establish a lexicon that will be used throughout my thesis. The former includes an examination of cryptography and secret splitting from a long-term perspective. The latter is important, as many security terms are heavily overloaded. Second, as I did with security, I provide a discussion of data deduplication, a popular technique for improving storage efficiency. Third, I summarize the design motivations that developed over the course of my study, and my conversations with long-term data custodians. The final section is a brief aside. It provides an abbreviated discussion on application preservation and long-range thinking. While this issue is largely orthogonal to the work I have addressed, it is worth mentioning and tends to arise in any discussion of long-term, data archiving.

2.1 Security Mechanisms

Storage security is a multifaceted area with many overloaded terms, and is too broad to be discussed completely within this thesis. The goal of this discussion, therefore, is to remove any ambiguity resulting from commonly used security terms, and to establish the role of

security in a long-term, archival perspective. To that end, the remainder of this section proceeds as follows. I begin with a short discussion on security mechanisms in general. This includes a discussion of mechanisms versus policies, and a summary of the categories used to describe how resistant security mechanisms are to attacks – an important factor to consider when evaluating the long-term security implications of a mechanism. Then, I provide some background information on one of the most common security mechanisms: encryption. Following that, I cover secret splitting, another common security mechanism.

In general, systems that claim to be secure enforce three primary policies [90]: secrecy, integrity and availability. While all of these are important properties, they do not represent the complete gamut of security policies. For example, various approaches have been developed to enforce anonymity [37, 133, 135, 180], plausible deniability [13, 37], accountability [120, 150, 151, 203], and more. That being said, the model that I am concerned with (secure, long- term archives) focuses primarily upon providing secrecy, integrity and availability for data with a potentially indefinite lifetime.

An important distinction to make is the difference between a security policy and a security mechanism. Policy is a description of a system's intended *behavior*. For example, a secrecy policy might state that data should only be readable by authorized users. In contrast, mechanisms are concerned with *implementation*. Continuing with the previous example, a secrecy policy could be implemented using encryption. Put another way, policies describe a system's security goals, while mechanisms are how those goals are achieved. This distinction is important to make, as policies for short-lived data can be very similar to policies concerning long-lived data. Unfortunately, problems often arise when the same mechanism is used for both short-lived and long-lived data.

A security mechanism's resistance to computation based attacks can be described using a four category framework [158] (summarized in Table 2.1). First, *computational security* states that a cryptosystem cannot be compromised in less than a specified number of steps. While many systems would seem to fall into this category, in truth, it is very difficult to definitively state that a system is computationally secure; a mechanism's resistance to different types of attacks can vary widely. It is important to note that computationally secure does not merely mean that it is computationally feasible to circumvent a mechanism, but also that it can be shown to be possible within a certain number of operations. Thus, an important distinction must be

Security Level	Description
Ad-hoc	argued to be secure without any rigorous evidence
Computational	secure against an opponent allowed to perform a *specific* number of operations
Provable	reducible to a well-known, intractable problem
Unconditional	secure even to an opponent with an unbounded amount of computation

Table 2.1: Classifications used to describe a security-mechanism's resistance to computation based attacks (ordered by resilience from weakest to strongest).

made between computationally secure and computationally *bound*. The later means that, given enough computational time, the mechanism can be compromised. This is a critical point when dealing with long-term security. The second category, *provable security*, reduces the attack to another, well-known problem. For example, a cryptosystem might be shown to be provably secure based on the difficulty of factoring large, prime numbers. Despite what the name might suggest, provable security is only a proof relative to another, usually intractable, computational problem. It is not the same as a proof that shows that an adversary with infinite computational time cannot compromise the mechanism. Third, *unconditional security* is secure in the face of an opponent with an unbounded amount of computation. This is the strongest security class. It implies that, even to an adversary with infinite computational time, it can be proven that the mechanism is secure. A fourth class, and by far the weakest, *ad-hoc security* is also mentioned on occasion. It describes mechanisms that are argued to be secure but for which no rigorous analysis can be made. So-called "security through obscurity" techniques often fall into this category. It should also be mentioned that almost nothing is completely secure. Even mechanisms that are unconditionally secure can fall prey to human level cryptanalysis techniques such as social engineering or physical coercion (so-called rubber hose cryptanalysis techniques).

While the ad-hoc security level may seem to describe mechanisms that are without value, quite the opposite is true. One such example, randomness, has become increasing popular. In the area of application security, considerable effort has gone into increasing an adversary's workload through the introduction of randomness [53]. Attacks that rely upon memory exploits are often based on the fact that information is located in very predictable and consistent locations. Randomization is a means of forcing an adversary to analyze each copy of the program they are attacking. This increased difficulty does not, however, makes the attack

impossible. Thus, the increased attack time can be combined with other strategies, such as signature-based attack detection [83, 182, 198]. However, an important difference between application protection and data protection is that a user's data is unique and personal. Thus, even one compromise could mean that unique data has been lost.

2.1.1 Encryption

Encryption is one of the most common mechanisms encountered in the area of computer security. The traditional introduction to encryption involves users, Alice and Bob, who wish to communicate in a way that they can understand each other while their fictional adversary, Oscar, cannot. Using a predetermined key, Alice encrypts her plaintext message into ciphertext. When Bob receives the message, he is able to decrypt the ciphertext back into plaintext– he also has the predetermined key and he knows the cryptosystem that Alice used in the original encoding. If Oscar obtained the ciphertext message, he would be unable to decrypt it because he does not have the key. An important observation to make is that it is assumed that Oscar knows the cryptosystem that Alice and Bob used. Kerckhoff's principle states that a cryptosystem's security comes from the adversary not knowing the encryption key; it is assumed that the adversary always knows what cryptosystem is being used [158].

With a few exceptions, most encryption algorithms are provably secure. In other words, the most secure that the cryptosystem can be reasoned to be is based on the difficulty of a related, intractable problem. For example, some public-key cryptosystems rely on the difficulty of factoring large numbers. While unconditionally secure cryptosystems exist, one-time-pad systems for example, they often incur a management cost that makes them unwieldy at best. As the security of encryption is based on upon the difficulty of a related problem, the struggle between cryptography and cryptanalysis can be viewed as an arms race. For example, a DES encrypted message was considered secure in 1976; just 23 years later, in 1999, the same DES message could be cracked in under a day [158]. From a long-range perspective, it is very difficult to predict the future of cryptanalysis. For example, advances such as quantum computing have the potential to make many modern cryptographic algorithms obsolete. While cryptography works reasonably well for short-term data secrecy, it does introduce a number of issues that become magnified as data lifetimes grow longer. At the extreme end, data with indefinite lifetimes (as is the case for long-term archival storage), these effects can be dramatic.

Cryptosystems are generally divided into two categories based on their key arrangement: symmetric-key and public-key. In a symmetric-key cryptosystem [152], the key that is used to encrypt the message is the same as the key used to decrypt the message. Thus, in the previous example, both Alice and Bob share the same symmetric-key. In contrast, public-key cryptosystems utilize two keys: a public-key and a private-key[151]. In this key arrangement, Alice encrypts messages for Bob using Bob's public key, and Bob decrypts messages using his private key.

While cryptosystems themselves can be described by their key arrangement (symmetric versus public-key), the use of cryptography can be described based on the lifetime of the ciphertext. In long-lived encryption, such as for an encrypted file in an archival storage system, the data is indefinitely persistent. In contrast, there are also short-lived uses of encryption. These uses may not suffer from the same pitfalls as long-lived encryption, owing to their low persistence requirements.

Examples of short-lived encryption include the use of cryptographic primitives in authentication and authorization policies. Many such systems rely upon the idea that key distribution, if carefully controlled, can imply a user's identity [116, 150]. This often requires the use of a trusted key generating authority with the ability to authenticate users. Further, this trust issue can be extended to include capability granting, in which the key generator also provides users with "tickets" that can be redeemed for services [92, 156]. In this model, the loss of an encryption key does not pose a great problem, as new tickets can be generated with relative ease.

Another example of short-lived encryption is session security. Some protocols, such as transport layer security (TLS) [44], are concerned with protecting data transmissions from eavesdroppers, replay attacks and message forging. Others provide anonymous communications using encryption as way of masking the source and destination [46, 133]. In these schemes, a message is routed through a pre-determined number of hosts. The sender shares a unique key pair with each host that the message will be routed through, and uses these keys to "wrap" the message in multiple layers of encryption. As the message is routed through the hosts, the receiving node decrypts their layer of the message and passes the message onto the next host. As with other short-lived uses of encryption, the loss of an encryption key is not catastrophic.

In contrast to short-lived encryption, key management in long-lived encryption is con-

siderably more problematic. With persistent ciphertext, a number of scenarios would motivate the need for eventual re-encryption. Examples include key rotations, compromised keys, compromised encryption algorithms and access revocation. In the very least, the computationally-bound nature of cryptographic security implies that, when data lifetimes are long enough, re-encryption is inevitable; processing power is monotonically increasing. In some cases, re-encryption may only be needed for a few files, but in other cases, many petabytes of data might need re-encryption.

Especially when required due to a key compromise or algorithm exploit, re-encryption of data must be done in a timely manner, However, the time to apply the new encryption technique to a large amount of data will probably not be the limiting factor; rather, issues with obtaining keys and reading the old data quickly may be more critical. Even this assumes that a custodian exists to oversee such a process. Further, optimizations to speed migration likely come with an associated management cost. For example, if a system chooses to save time by encrypting over the old algorithm, it must have a way of dealing with key histories and key distribution. With long data lifetimes this becomes increasingly complicated, as key histories would need to be preserved. In contrast, if the system chooses to decrypt the data before applying the new algorithm, then it must have access to the users' encryption keys. Both scenarios must also take into account the threat from malicious inside attackers. If an insider has access to a user's encryption keys, the security of the system is intrinsically weakened; encryption as a security mechanism relies on carefully controlling access to keys.

Another problem introduced by the long-term usage of encryption is the threat of key loss. In an archival storage system, data can be very difficult to reproduce; the software, hardware and even users that produced the data may no longer be available. Encryption keys are a single point of failure and key loss is effectively equivalent to short-term data deletion. Unfortunately, it is not equivalent to secure data deletion. The loss of the encryption key renders the data temporarily unavailable. While the cryptosystem is valid, access times are negatively (and drastically) impacted. Unfortunately, due to the computationally-bound nature of cryptography, the data may be readable in time, and is therefore not securely deleted. To mitigate the problem introduced by key loss, many systems that utilize encryption include a key-management aspect. Unfortunately, this often introduces an implicitly trusted key authority, which in turn increases the risk of a malicious inside attack.

The fear with any long-term use of cryptography is that an attacker with enough time and computational resources can compromise any policy enforced using encryption. Numerous examples exist of archives being lost or misplaced [25, 108, 109, 123, 201]. In such a scenario, a malicious user that obtains encrypted data would merely need to wait for cryptanalysis techniques to catch up to the cryptosystem used to protect the data. Thus, it must be assumed that an adversary with enough computational time can circumvent any policy that utilizes cryptographic primitives. It is, therefore, fair to say that encryption only provides adequate security when the relevant lifetime of the data is shorter than the time needed to compromise the cryptosystem.

2.1.2 Secret Splitting

While keyed cryptography is the mechanism most often associated with data secrecy, another popular mechanism is secret splitting [33, 74, 124, 129, 130, 195], in which a secret is distributed to a number of share holders. The unconditionally secure nature of some secret splitting schemes suggested from a very early stage that they might be well suited to long-term security [163, 166].

There are two general classes of secret splitting techniques: n of n schemes and m of n threshold schemes. Each of these classes is comprised of a number of different algorithms that differ in their security, performance and feature set. Both techniques produce a set of n *shares* from the data to be kept secret. With an n of n scheme, all n of the shares are required in order to reconstruct the data. In contrast, with m of n threshold schemes, the secret is used to generate n pieces, any $m \leq n$ of which can be used for reconstruction; both n and m are determined at the time of splitting [153].

The classic, if not slightly morbid, example of secret splitting is the distribution of missile launch codes. In this scenario, the goal is to distribute the weapons' launch codes to a number of trustees, such that a quorum of shareholders must agree to provide their share in order for the missiles to be launched. An m of n scheme could allow the launch of the weapons even if some of the shareholders are either unavailable or unwilling to provide their share of the launch code [1].

The performance of different secret splitting algorithms varies widely. The binary XOR operation, a computationally inexpensive operation, forms the basis of a relatively straight-

[1] According to a *Time Magazine* article, control of Russian nuclear weapons relied upon a 2 of 3 threshold scheme with secret shares distributed to the President, Defense Minister and the Defense Ministry [157]

Figure 2.1: A simple *n* of *n* XOR based secret splitting scheme in which a secret, *S*, is split into *n* pieces all of which must be recombined in order to reconstruct *S*. In this algorithm R_1 through R_{n-1} and S' are distributed amongst the secret shareholders.

forward technique. Illustrated in Figure 2.1, it is fast enough to provide security in systems which must provide relatively low-latency data access [13, 169]. The *n* of *n* XOR-based algorithm operates as follows:

1. Randomly generate $n-1$ pieces of data (R_1 through R_{n-1}) equal in size to *S* (the secret to split)

2. XOR R_1 through R_{n-1} and *S* to produce S'. ($R_1 \oplus R_2 \oplus \ldots \oplus R_{n-1} \oplus S = S'$)

3. Securely dispose of *S* and distribute R_1 through R_{n-1} and S' to the secret share trustees

While the simple *n* of *n* XOR approach is relatively fast, other approaches rely on far more expensive operations, and thus, are considerably slower. For example, Shamir's original *m* of *n* threshold scheme relies on linear interpolation and proceeds as follows. The secret *S* is divided into shares using an $m-1$ degree polynomial $q(x) = S + a_1 x + \ldots + a_{m-1}x^{m-1}$ where a_i is randomly generated. The first share is generated by calculating $q(1)$, the second by calculating $q(2)$, and so forth up to $q(n)$. Thus, it stands to reason that any *m* of these values, and their index, is sufficient to deduce the value of *S*, while less than *m* reveals no information. However, as linear interpolation is more expensive than a binary XOR operation, Shamir's threshold scheme is slower than the *n* of *n* XOR based scheme.

A useful characteristic of some secret splitting schemes is perfect secrecy. In such schemes, it can be proven that combining any fewer than the pre-determined number of shares, $m \leq n$, reveals *no* information about the original secret. For example, as Figure 2.2 shows, using the XOR based algorithm, with less than *n* shares, all possible values of *S* are all equally likely. This unconditional secrecy, as described in Table 2.1, is fundamentally different than the computationally-bound security provided by most keyed cryptosystems. Thus, unconditionally secure secret splitting schemes can provide rigorously provable, future-proof security; the

	Owners	Adversary
R_1	1100	1100
R_2	1010	1010
R_3	1110	1110
R_4	0101	????
S	1101	????

Figure 2.2: Example of the unconditionally secure nature of an XOR based secret splitting algorithm with perfect secrecy ($S = R_1 \oplus \ldots \oplus R_4$). Even with three of the four secret shares (R_1 – R_3), the adversary is no closer to S because all values of R_4 are equally likely (assuming that R_i are all randomly generated). Thus all values of S are equally likely as well.

inevitable need for re-encryption in largely avoided. Further, secret splitting is well suited to long-term archival usage, since the slower speeds of many secret sharing schemes, compared to encryption, are not a major detriment in archival workloads.

While displaying several useful characteristics, it is important not to assume that secret splitting is a cure-all panacea for the shortcomings of long-lived encryption, or that it can be effectively used as a drop-in replacement for cryptography. In any system that utilizes secret splitting, there are several concerns that must be addressed.

One problem with secret splitting is ensuring that a malicious adversary, who is able to obtain one share, cannot easily determine and find the other shares needed for reconstruction. This is especially true when guarding against malicious insiders. With cryptography, a system can store encrypted data and, without the corresponding encryption key, there is a relatively low chance that the data will be readable by an unauthorized user in the short-term. If the key is not stored with the data, even the abilities of a malicious insider are mitigated. However, with secret splitting, a malicious insider, with knowledge of which shares reconstruct the data, and where those shares are stored, can easily launch a targeted attack and obtain the data.

As with keyed encryption, secret splitting must takes precautions against the loss of key material. Knowing which secret shares to combine is analogous to an encryption key; it is the secret that transforms ciphertext into plaintext. Recovering data from secret shares is a difficult problem in the absence of any assistance, as the naïve approach would involve testing every possible combination of shares. Thus, the use of secret splitting must include a secure

recovery mechanism that does not inadvertently empower an attacker (inside or outside the system).

Storage overhead is another problem with secret splitting. In both *n* of *n* and *m* of *n* algorithms, storage blow-up is typically on the order of *n* times. Some schemes require less storage overhead [129], but in return offer less than unconditionally secure secrecy guarantees. This is in stark contrast to encryption, which incurs minimal storage overhead. The increased storage requirements imposed by secret splitting provide further evidence of the need for economically efficient, archival storage.

Secret splitting is well suited for long-term, archival storage for a number of reasons. First, it is unconditionally secure, and thus can provide long-term security. Second, threshold schemes do not suffer from the single point of failure that encryption introduces. Finally, archival workloads can accommodate the computationally expensive, and therefore slower speeds, of many secret splitting algorithms. They are not, however, a drop in replacement for encryption, and require designs that take in account the need for data recovery and the threat of malicious insiders.

2.2 Single Instance Storage

A common technique found in some archival storage systems is content-based naming [69, 127, 205, 206]. This technique uses a hash of the data as the data's name and offers a number of benefits. First, content-based naming simplifies data-reliability checking as the name of the data provides an easy way to check data integrity. Second, content-based naming has been utilized as a means of reducing storage overhead; if multiple users posses the same file, a storage server would only need to store that file once.

Taking single instance storage a step further, data deduplication can be applied not only at the level of entire files, but can also be used to identify matching blocks within files. The first intra-file technique is exemplified by the Venti archival storage system [127]. In Venti, files are broken into fixed sized blocks before deduplication, so files that share some identical contents (but not all), may still yield storage savings. The second, and most flexible form, breaks files into variable-length "chunks" using a hash value on a sliding window; by using techniques such as Rabin fingerprints [128], chunking can be done very efficiently. This technique has been used in a variety of communications and storage systems [14, 113, 202].

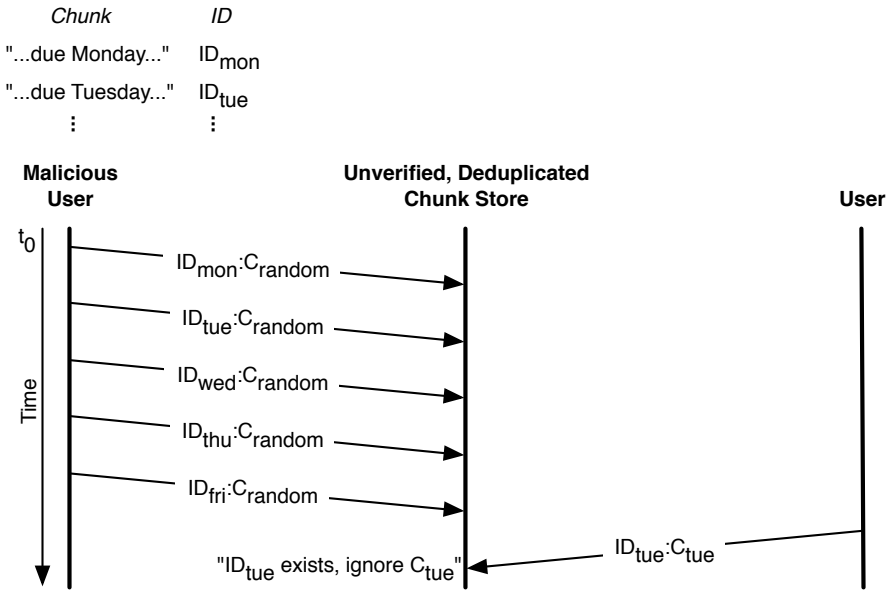


Figure 2.3: Targeted-collision attack in which a malicious user exploits predictable data (in this example, a form letter with a due date) to generate valid chunk IDs, and associate those IDs with invalid chunks. If the user is the first to submit the ID, subsequent chunks will be deduplicated to a garbage value.

Unfortunately, conflicts can arise when attempting to utilize deduplication techniques in a secure storage system. Deduplication takes advantage of data similarity in order to achieve a reduction in storage space. In contrast, the goal of cryptography and other secrecy mechanisms is to make ciphertext indistinguishable from theoretically random data. Thus, the goal of a secure deduplication system is to provide data security, against both inside and outside adversaries, without compromising the space efficiency achievable through single-instance storage techniques.

One strategy for combining deduplication with security is convergent encryption [48, 161]. This technique uses a function of the hash of the plaintext of a chunk as the encryption key: any client encrypting a given chunk will use the same key to do so, so identical plaintext values will encrypt to identical ciphertext values, regardless of who encrypts them. While this technique does leak knowledge that a particular ciphertext, and thus plaintext, already exists, an adversary with no knowledge of the plaintext cannot deduce the key from the encrypted chunk.

In addition to the difficulty of combining security mechanisms with deduplication, single instance storage opens up the possibility of targeted collision attacks. In a deduplicated

chunk store, a targeted-collision attack could be used to associate a false value with a given key. The pivotal difference between random collisions and targeted collisions is that a user can exploit the predictable content of some data — in Fig 2.3 the malicious user utilizes similarities in form letters — to generate valid chunk identifiers. If an adversary can be the first to submit those identifiers with a garbage chunk, and if the chunk store cannot verify the correctness of the identifiers, subsequent submissions that have the same identifier will be deduplicated to the garbage chunk.

While content based naming, and more generally deduplication, may be useful for increasing a system's storage efficiency, it does raise a number of concerns in long-term archival storage. First, while the mean time to data loss is not affected by deduplication, multiple files may be compromised from the loss of a single chunk. Second, it does add another level of complexity, as retrieving a file now relies upon two distinct actions: identifying the blocks to retrieve, and locating each of the respective blocks. Third, as discussed, it can complicate the integration of other desired storage properties, such as security. Finally, as the next section discusses, it may remove a level of transparency that some custodians of long-term data prefer.

2.3 Archival Design Guidelines

During the course of the research for this thesis, and through discussions with long-term data custodians, a number of design motivations and archival needs have become increasing evident. Of course, as part of the challenge of building effective long-term storage systems, finding the right compromise between conflicting needs can be difficult. This section identifies some of the core directives that should guide the design of a storage systems intended for long-term data.

2.3.1 Capacity Scalability

Archival storage is well served by a scalable storage system. More specifically, this relates to two facets of scalability: granularity, and scale-out potential. The first, granularity, describes the size of each scale-out unit. The second, scale-out potential, describes the upper limits that a system can scale to before requiring a wholesale system replacement.

Capacity scalability is especially important in an archival storage scenario, because

accurate storage-need forecasts can be notoriously difficult to calculate; quite often, once the value of archival data has been recognized, custodians discover that they have underestimated the amount of data that is worth preserving [103]. This scenario is further exacerbated by the growth in file sizes. In addition, an effective archival system should present a low barrier to entry. This is important, as a long-term storage system should be able to grow with a data producer.

2.3.2 Efficiency

In keeping with the goal of enabling the preservation of all data that *might* one day be useful, archival storage must be concerned with cost efficiency. Put more succinctly, archival storage must be inexpensive. Further, since efficiency gains are relatively low when considered as an afterthought, cost efficiency must be a central goal from the very beginning of the design process. In this respect, its analog can be seen in computer security; while early system security was largely a tacked-on afterthought, it is widely held today that security must be present as a motivating design directive in order to be most effective [12]. Of course cost efficiency is not a discrete value, but rather the aggregation of a number of distinct design decisions.

A common approach to controlling costs is the use of commodity hardware [59, 68, 69]. Often times, the assumption in such a system is that higher failure rates are tolerable, or that the marginal cost of performance from enterprise class hardware is either unneeded or too costly. It is important to note that the cost savings yielded in such a strategy may be offset by an increase in management costs. For example, a system that expects higher failure rates from consumer class equipment may not factor in the additional administrator workload incurred from replacing these more frequent failures.

Administrative efficiency describes the effort to manage a system. Part of what makes this particular facet of efficiency difficult to measure is the scope of what it includes [11]. The management costs of a system should entail everything from initial planning to (in a traditional system) the end of life decommissioning of a system. Thus, administrative efficiency covers everything from the ease of integrating a system into an existing storage hierarchy, to the industrial design of a system's components. For example, in a large storage system, a large amount of an administrators job is replacing failed drives. A design that makes it difficult to locate and replace failed drives can drastically reduce administrative efficiency.

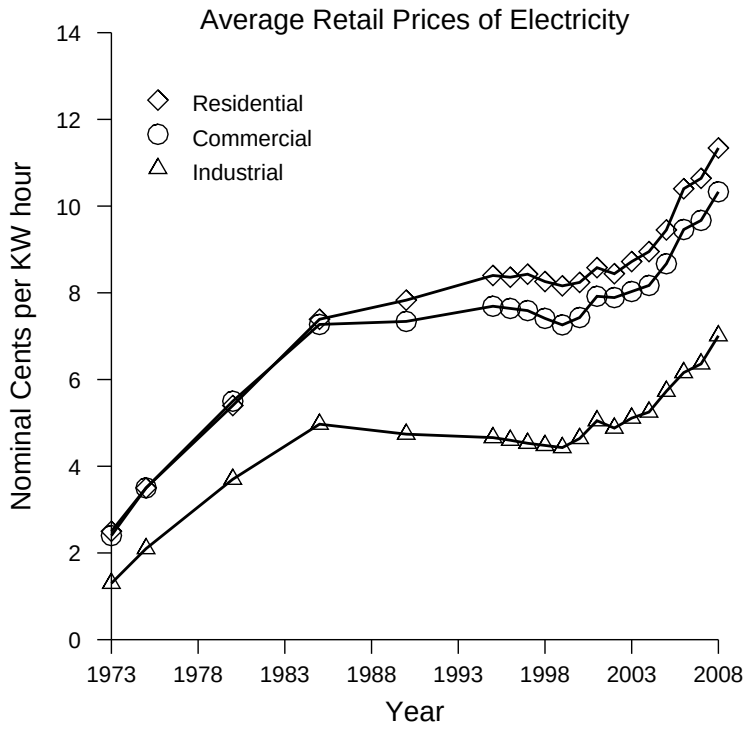


Figure 2.4: The nominal price of energy from 1973 to 2008 for three markets: residential, commercial, and industrial.

Recently, energy efficiency is an area that has been the focus of quite a bit of attention [121, 122, 188, 208]. In storage, energy efficiency is made up of two primary components. First, there is the energy to actually power the system itself. Second, there is the associated cost of cooling the equipment; as modern power supplies are not 100% efficient — typical power supplies are often only 65–75% efficient — they generate enormous amount of excess heat.

Unfortunately, most systems fail to address the opportunity costs associated with energy efficiency. In traditional storage systems this is not a major problem, since most systems will only be used for a few years. However, in a system that is to evolve over time, the concept of *energy inflation* must be addressed. As Figure 2.4 illustrates, the price of energy is increasing [51]. Further, as Figure 2.5 shows, hard drive capacities have been growing at an exponential rate [1]. In fact, Kryder's Law states that hard-drive capacities are increasing even faster than processor speeds; since 1956, hard drive capacity has increased 50-million fold [181]. Thus, there is a huge opportunity cost associated with aging hard drives. Combined, these two factors mean that every dollar spent on energy returns a monotonically decreasing amount of utility as measured by the storage capacity. In archival storage, the goal of an evolvable system means that it is insufficient to view energy efficiency as a static target. Rather, it must be dealt with proactively. Put more succinctly, if a system is not getting *more* efficient, energy inflation means that it is getting *less* efficient.

Storage efficiency is a measurement of how much usable storage is yielded from a given amount of raw storage. For example, reliability schemes that create a single mirrored copy of data reduce storage efficiency by half. Thus, for archival storage, the goal is to maximize storage efficiency without unduly sacrificing required levels of reliability.

2.3.3 Evolvability

Current storage systems tend to operate with a hardware lifespan of approximately five years. At the end of this lifespan, data is migrated to a new system, and the old system is replaced in what is sometimes called a "fork-lift" upgrade. Inadequate performance levels, increasing failure levels, or equipment end of life accounting often motivates this upgrade. Unfortunately, with its ever increasing corpus of data, such upgrades conspire against the cost efficiency needs of archival storage; such upgrades often incur a high cost in energy and administrator time.

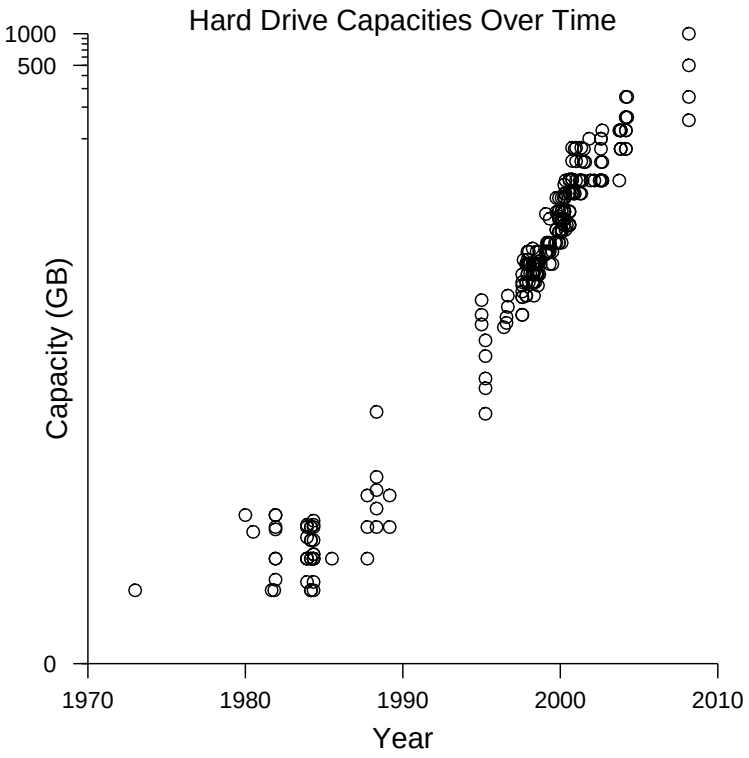


Figure 2.5: Hard drive capacities from 1973 to 2008.

Thus, archival solutions should be designed to scale across time [18, 19]. This has been elaborated as the ability to scale across changes such as technology, protocols, and vendors. One approach to achieving this evolvability is to abstract away the details of technology specific design, and export a very simple protocol, such as file level `put` and `get` methods; abstraction layers can map complex protocols to a set of fundamental operations. Further, this approach forces designers to call into question assumptions associated with an implicit, wholesale system migration. For example, in a long-term evolvable system, a drive cannot be left to participate until it fails; the amount of capacity an old drive provides for the power it consumes does not justify its existence.

2.3.4 Reliability

By its nature, the contents of archival storage can be very difficult to replace. Often times the people, knowledge, systems or input needed to recreate data may no longer exist. In other scenarios, such as with scientific computing and modeling, the time required to regenerate data may be prohibitive. Therefore, archival storage must be reliable.

At times, archival systems are called into use as a holding area for cold data before it is either relocated, or deleted. Ironically, in this scenario, archival storage must be reliable enough to safely store data until a custodian has had the opportunity to review its contents and potentially delete it. In such a situation, all data must be assumed to be irreplaceable until stated otherwise.

2.3.5 Reasonable Performance

The workloads of archival storage allow it to relax the need for ultra-low latency access if it returns gains in other areas. For example, a spun down disk results in energy savings, but may incur a spin-up fee of a few seconds. This is not to say, however, that random access performance is not important to archival storage. Continuing the example, a penalty of a few seconds to spin up a disk may be acceptable, but the scale of a few minutes to locate data on tape is an unacceptable delay. The key insight is that random access performance can be relaxed, not ignored, *if* it results in useful gains elsewhere in the system.

2.3.6 Transparency

A design property that is rarely identified in traditional near-line storage, yet often comes up in regards to archival storage, is that of transparency. In an ideal system, an ever increasing corpus of static data would eventually reside on a slowly changing, evolvable system. This, however, introduces data vulnerability in the form of vendor dependence; with storage higher in the hierarchy, lower level backups and replication can provide backup access. Thus, archival system should provide simple file level access methods that can serve as a worst-case exit strategy.

The issue of transparency often comes up in the discussion of techniques that operate at a sub-file level. For example, data deduplication using file chunks may work against transparency if chunks are distributed across many devices. Thus, while such an approach may increase storage efficiency, the implicit metadata and mapping involved with translating a chunk-level data view to a file-level data view may be unacceptable to many data custodians.

2.4 Application Preservation

Projects in the area of long-term data preservation often fall into one of two categories: data preservation and application preservation. Data preservation, the area that my work is concerned with, is about maintaining bits. In contrast, application preservation is concerned with making sense of the interpreted meaning of those bits [22, 63]. While application preservation is orthogonal to the scope of my work, it is an interesting problem and inevitably comes in up in any discussion of long-term storage. Therefore, the goal of this section is to provide a basic understanding of application preservation issues.

As mentioned earlier, long-term preservation of digital data is not a purely technical problem, and is poorly served by purely technical solutions. Considerable work, therefore, has gone into the human and organization level challenges. For example, The National Archives of Australia has made a concerted effort to detail a series of best practices to ensure that migration and digital custodial work can proceed as effectively as possible [75]. Part of this effort involved establishing a clear point of responsibility to replace the mostly ad hoc efforts spread across disparate organizations. More interestingly, their work identifies the break that digital records represent compared to physical artifacts; originality is no longer the issue, rather the capture and

playback of a temporal "performance" of a digital artifact is the aim. Specifically, they identify a three stage model of digital preservation: the *source* data, the combination of hardware and software comprising the *process* needed to interpret the source, and the actual *performance* that is rendered to the screen.

Efforts to preserve the process and performance aspects of application preservation generally fall into one of two camps: migration and emulation. The first, migration, involves the effort of data custodians that perform audits to detect endangered formats that can be translated to safer formats. To this end, work such as the PDF/A specification has been focused on producing formats that are well suited to long-term data [134]. The PDF/A specification is a subset of the complete PDF specification that limits features to those that can be reliably included in a self-contained, self-describing PDF file. The second, emulation, maintains the original source and performance components, and utilizes one or more virtualized environments for the performance phase [63].

At the current time, one of the critical aspects of application preservation is saving descriptive metadata. To this end, Howard Besser's advice is to, "save any metadata that is cheap and easy to capture" [22]. Further, metadata preservation has a number of important implications to preservation chores ranging from searching for data within bodies of long-term data, to assisting in data and format migration. Ironically, while metadata plays an important role in preserving long-term archival data, it also complicates long-term integrity; it is not enough to simply preserve data, metadata and the connection between metadata and data must also be preserved.

Within the context of my work, Chapter 5 discusses the design of Pergamum, which separates the internal storage of metadata and data, while encapsulating it within a single device. This is done for a number of reasons. First, in an effort to be cost efficient, the unique needs of metadata and data — metadata currently has lower capacity needs, but is accessed more frequently than data — are addressed with two different media types; archival storage cannot afford the luxury of excess, therefore the scale of the solution must match the scale of the issue. Second, while internally data and metadata are treated separately, externally they are unified into one field replaceable unit. Third, the needs of data and metadata may evolve along separate paths; while the total capacity needed for data eclipses metadata, some predict that eventually metadata may one day be larger than the data itself. By establishing a strategy of treating the

two as unique, it is much easier to adapt newer devices to their changing needs.

Taking a step back from the specifics of computer storage, there are a number of on-going projects in the area of long-term thought and design. Two of the most famous are the Clock and Library of the Long Now [24]. Both endeavors are key projects of the Long Now Foundation, and illustrate the group's 10,000 year view of the future. Their philosophy of "slower is better" serves as a counterpoint to what they see as the current "faster and cheaper" mentality.

Chapter 3

Related Work

> The moments of the past do not remain still; they retain in our memory the motion which drew them towards the future, towards a future which has itself become the past, and draw us on in their train.
>
> Marcel Proust

The current corpus of storage research consists of a wide gamut of systems and storage models. As with many terms in computer science, even the narrower scope of "archival storage" is overloaded and involves numerous, sometimes conflicting, models. For example, public archival systems are often built expressively to ensure open access; library-like systems are designed to ensure equal access to all potential readers. In contrast, other systems are designed specifically to provide data secrecy.

The disparate needs of long-term preservation, and traditional, performance-oriented storage, warrant a solution specifically designed for archival storage's workload, access model and data lifetimes. My work is important because it directly addresses archival storage security, energy-efficiency and evolvability. To that end, the two goals of this section are to present my work in the context of existing research, and to demonstrate that no current systems fully addresses the needs of long-term, secure storage.

The remainder of the chapter consists of an overview of several important areas of relevant research. I begin by describing the trend towards domain specific systems. That is, storage systems designed for very specific workloads and access patterns. Second, as distributed architectures are well-suited to evolvable storage, I discuss the intentions and designs of several

prominent, distributed storage systems. Third, I present an overview of systems designed for a wide variety of security properties. Finally, I discuss several approaches that have been taken to address the high costs of data storage.

3.1 Workload Specific Storage

A growing trend in storage systems is the development of systems designed with very specific workloads in mind. Often times, these domain or application specific systems are optimized for a unique access patterns or requirements. While this presents the challenge of how best to optimize for key operations, it also presents opportunities. Domain specific systems are often able to relax constraints that would prevent them from being suitable as a general purpose system. For example, in archival storage, reasonable access latency penalties may be an acceptable trade-off for gains in energy efficiency.

The purpose of the archival model that my research is concerned with is to preserve relatively static data that has been purposely prepared for long-term storage. This class of data tends to exhibit a write-once, read maybe workload. In a business setting this would be data such as documents that have been finalized and require retention. This is becoming increasingly important as legislation increases the retention and recovery demands of such data [2, 3]. In a personal setting this would include such family history information as legal documents, medical records, images, videos and correspondences.

As the contents of a secure, archival storage system are relatively static, I am concerned with a position that is adjacent to the traditional storage hierarchy [76, 194]. During data's production life cycle, content destined for long-term archiving exists within the storage hierarchy. In this phase of its life, it warms and cools and thus ebbs and flows between the gradation's layers. However, upon reaching its final static form, it is prepared for long-term preservation. Part of this procedure involves migrating the data out of the hierarchy and into the class of archival storage system that my work is concerned with.

Finally, the protection goal of long-term secure storage is to provide secrecy over the data's entire, potentially indefinite, lifetime. More over, the security approach must effectively balance secrecy with availability. On one hand, the data secrecy must be sufficiently secure that it does not require attention from a data shepherd. However, it must simultaneously be flexible enough to be readable by an authorized viewer with no outside information about the data.

GPFS is a file system designed for high performance cluster-based computing [146]. These installations are often used in domains such as scientific computing, where workloads are marked by large numbers of concurrent accesses to the same file (although not necessarily in the same location within the file). To deal with this concurrent heavy workload, GPFS divides each file into equal sized blocks and these blocks are distributed on different disks within the storage environment. This allows multiple clients to hold locks on different portions of the file. This distributed locking allows the file system much greater efficiency for parallel access. Distributed locking is based on a central lock manager that distributes capability tokens to clients. Subsequent accesses to the data do not require further contact with the lock server. Allowing clients to lock a specific byte-range as opposed to a whole-file, or even whole-block lock further optimizes locking. In the write once, read maybe workloads of archival storage, such heavy-weight locking strategies are unnecessary, and only add unneeded complexity.

The author's of Slash [117] are concerned with the data starvation problems in large, HPC environments. Their view of archival storage is that of tertiary storage. The high data production rates found in HPC workloads must provide cost effective storage for large volumes of data, while still supplying data consumers in an efficient manner. Slash is an abstraction layer between low level, archival storage and higher level storage. Their solution is composed of metadata servers, a cache component and an archiving component. The system has two goals. First, it acts as a cache for storage higher in the hierarchy. Second, it acts a gateway between the archival system and the upper level, HPC compute system.

Dynamo is a distributed key-value store that Amazon relies upon for many of their internal applications [42]. The system provides "always-on" write reliability and favors availability over consistency. Making this distinct from archival storage is the system's insistence on low latency SLA's, and fairly high levels of administrator input for certain operations. However, similar to archival storage is the need for incremental scalability, non-specialized peers, and decentralization. One area of interest is that this system utilizes "hinted handoffs" which are similar to the idea of foster writes [114].

The Google File System [28, 59] is another example of a file system that was made for optimal performance under a very specific workload. In this particular case the workload was marked by a number of very large files that are rarely deleted, and where most mutations come in the form of appends rather than writes to the middle of the file. The basic design of the

Google File System involves a number of storage hosts and a central master server. Files are divided into fixed sized chunks throughout the system and replicated a fixed number of times in order to adequately deal with failures within the system. This strategy works as the system are designed with the specific file workload in mind. Other, more general purpose storage systems have specifically provided flexible redundancy schemes that allow files to specify their desired level of reliability [4, 190]. In GFS, Client requests are sent to the master server and include the file name and the calculated chunk index. In an archival scenario, this presents a number of problems. First, the master servers present a centralized point of control that introduces a point of failure, and complicates evolution. Second, the system's fixed replication rate is suitable to a workload such as Google's that has no strict permanence guarantees, but would be inadequate for the long-term survivability requires of archival data.

FAWN (Fast Array of Wimpy Nodes) was created as power aware system for a workload dominated by seek-dominated key, value storage [10]. While this workload is almost the exact opposite of the write dominated archival space, this is the sort of workload that a DNS server, or other look-up service might encounter. Their solution involves a number of relatively high powered front-end systems, with an array of low-powered, DRAM based devices for storage. Unfortunately, the current design of FAWN does not address the reliability problems associated with flash outside of straight replication. Additionally, static costs are not addressed by their system, and their reliance on flash suggests that this might be rather high. Additionally, the front-end nodes are fully aware of each node in the system, and are updated on each node entrance and exist (it is, therefore, not a fully distributed system). They do, however, utilize a Chord like DHT arrangement. Although, as stated earlier, fast lookup times rely heavily on caching and a front-end system that can route the queries directly to the correct node.

3.2 Distributed Storage

Distributed systems, including storage systems, have been an active area of research for some time. Designs that avoid a monolithic architecture in favor of a decentralized approach have been applied to a wide range of pursuits including resource utilization, survivability, and performance. Archival storage is well suited to a decentralized architecture, especially when the interface between components is designed to facilitate graceful system evolution. Of course, as with any fundamental design shift, opportunity comes with challenges to overcome.

As even moderately powerful computer systems often have local storage that goes underutilized, attempts have been made to aggregate this distributed resource and present it as a single pool of storage [36]. One such example, Frangipani was designed for trusted networks and utilizes a layered approach [175]. At the lowest layer are the physical disks. Above this is Petal, which presents multiple physical disks as distributed virtual disks [91]. Frangipani is layered atop this, and presents views of virtual "file servers". Frangipani is designed to be used on a cluster of machines under a common administrator. Thus, while block-level secrecy is important, the system's security primarily stems from trusted operating systems and secure communications channels.

Other attempts to utilize contributed storage widened the source of participating systems, forcing designers to take a harder look at system security; on a public system, such as the Internet, users may not know which nodes their data will resides upon [7, 81]. These systems, therefore, often utilize secrecy through encryption. For example, the authors of PAST [143] explicitly assume that it is computationally infeasible to break their encryption. Often times, as in OceanStore [86, 137], the system itself does not directly address the use of encryption beyond stating that data entering the system must be encrypted. In such a strategy, the onus introduced by encryption — such as key management, re-encryption, and key rotations — is left to the user or outside solutions.

Another use of contributed storage, and a useful technique well suited to archival storage, is the use of geographic diversity to increase data survivability [82, 145, 186]. One example of this approach, Glacier [70], utilizes extensive use of erasure codes and redundant distribution. The central idea is that high levels of redundancy provide high levels of reliability. While the approach does incur a high storage cost, the authors mitigate the problem by aggregating smaller objects and utilizing garbage collection. Lastly, while such geographic diversity can introduce latency delays, Glacier, like other such systems, is not intended to be a primary store. Instead is exists alongside a primary store that users utilize for low-latency access while Glacier is used for long-term accessibility.

An accessory to geographic disparity, data encoding if often utilized in place of straight replication. Stonebraker and Schloss introduced distributed RAID [160] to provide redundancy against site failure via geographic distribution and RAID-style algorithms. This technique was further refined by Myriad [29], which uses a logical disk abstraction in conjunc-

tion with groups of data blocks and parity blocks that are produced using erasure codes. While such cross-site redundancy strategies do introduce an additional level of management overhead, studies show that cross-site redundancy techniques add considerably to data reliability, even when less than optimal encoding techniques are utilized [29].

Outside of extensibility, distributed storage enables high degrees of parallelism; instead of a single, monolithic storage controller, requests can be routed to any number of storage hosts [26, 41]. Taking the idea a step further, several systems utilize dedicated metadata nodes and storage nodes, further reducing potential bottlenecks [110, 118, 189, 190]. For example, Ceph is comprised of intelligent object storage devices (OSD), a cluster of metadata servers, and clients. Scalability is provided by placement groups, and a function which provides inode to block mapping (as opposed to a static table) [77].

Intermemory exploits distributed systems for both reliability and lower access time [31]. Intermemory's design was created based on the report by the task force on archiving of digital information. Intermemory implements a block level substrate that can be used to build larger more complex data structures. They use two levels of splitting, which is similar to POTSHARDS, but each level relies on an IDA, as there is no explicit need for secrecy. The reason that Intermemory does two levels is to reduce the amount of network connections needed for a rebuild. It is unclear however if the splitting parameters at the two levels or fixed or tunable. Unfortunately, while the increased fan-out achieved with two levels of splitting can benefit access patterns, it can also result in significant management overhead. As it stands in Intermemory there is a lot of mappings that needed to managed.

Of course, while contributed storage can offer a number of benefits, it can also introduce problems, such as ensuring that participants are well behaved [20]. This is especially important in open, communal arrangements where each participating node acts as both a client and a storage node. One approach to the free rider problem, where a node does not actually fulfill its storage obligations, is the use of periodic requests and the looming threat of the end of a mutually beneficial relationship [85, 95]. Another approach relies on mathematical properties of erasure coded storage [73, 148]. The key insight into this approach is that the signature of the parity is the same as the parity of the signatures. While these systems can provide short-term data reliability through replication and geographic disparity, such an approach on its own is ill suited to long-term storage. These systems provide no guarantees about the persistence of each

replica, and often require a user to take a rather active role in ensuring the proper behavior of their storage hosts.

Similarly PAST is a peer-to-peer model that attempts to fairly balance the storage demands on each node in the system [143]. The system is arranged as an overlay network and uses Pastry [142] as a lower level that handles routing chores. One of the novel aspects of PAST is its methods for normalizing the storage usage on member nodes. Names are evenly distributed and identifiers are created by hashing the file's name, the owner's public key and a randomly chosen salt. PAST does utilize smart cards to manage the symmetric key operations. A novel aspect of PAST is the effort that has been put into storage management. This is understandable based on the fact that it is hinted that PAST would be a "pay for play" sort of system. Storage management is handled in two key ways: replica and file diversion. Replica diversion allows a node that is not one of the k numerically closest drives to store a file in response to a node without the necessary storage space. File diversion occurs when all of a node's leaf set is reaching capacity.

3.2.1 Distributed Communication

While distributed architectures offer a number of benefits over monolithic architectures, they also introduce a number of problems. As the system must deal with numerous, often transient, nodes, communication in distributed systems is a rich area of research.

As distributed systems are composed of loosely coupled, independent devices, system wide knowledge can be challenging. An extreme approach is the pursuit of global awareness, in which a fully connected graph allows one-hop communications between any two nodes. For example, in Name-Dropper, the system converges on global knowledge by having nodes randomly sending a neighbor a list of all the nodes it knows about [72]. The receiving node then adds these to the list of nodes that it knows about. Unfortunately, the per node storage overhead, and proliferation of messages with these strategies make them suitable for only relatively small systems.

Another problem with systems that attempt to achieve global knowledge is the difficulty of determining a termination point. The work of Kutten, Peleg and Vishkin relies upon either knowledge of the total number of nodes in the system, or by knowing the maximum total number of nodes (perhaps based on the naming techniques capacity) [87]. Their solution works

by transforming tall trees of links into stars, and then merging stars into a tree. The work of Abraham and Dolev does not require prior knowledge of the system's size [6]. Their approach involves the search for a leader node. As each node enters the system, it assumes that it is the leader, and then iteratively contacts other nodes in an attempt to either merge leadership or assume leadership.

Unlike systems that attempt to maintain a full list of network membership on each node, a number of approaches have shown that adequate coverage can be achieved through the use of randomization. The "small-world" phenomenon demonstrates that a k-nearest neighbor approach with even small amounts of randomization nets has a profound reduction of hops in peer to peer message routing [184]. Based on this behavior, SCAMP only requires each node to maintain a partial membership list [55]. The SCAMP approach involves subscription requests that can be forwarded along a randomly generated path, but cannot be dropped. The authors show through simulation that this process has coverage rivaling a system that requires each node to maintain a full membership list.

Related to the problem of global knowledge, is the problem of global consensus [88, 89]. Chlebus and Kowalski explored the use of gossip with the goal of getting large groups of nodes to agree on a common value [32, 187]. Their solution involves nodes maintaining and trading arrays of information through the use of collectors and disseminators. The problem is the huge amount of information that is transmitted. The authors use two metrics to measure their solution: time and the number of point to point messages. Unfortunately, in an array of n nodes, each node maintains three arrays of length n. Further, all three of these arrays are transmitted as a part of the update process.

Further work has extended gossip based communication approaches, making it Byzantine fault tolerant [89, 93]. The goal of this solution is to be stable in the face of Byzantine failures as well as resistant to free riders that do not act out of altruism (they only work for their own benefit). The solution is based on pseudo-randomness that attempts to take advantage of the benefits that randomness leads to gossip protocols, while still functioning in a deterministic fashion. Additionally, they use public-key encryption to enforce accountability on the basis that proof of misbehavior is a sufficient deterrent. This, along with delayed gratification, ensures that clients stick to established protocols.

In contrast to the lofty goals of global knowledge, a distributed hash table (DHT) is

a distributed data structure that maps keys to nodes. Given a key, these algorithms can, with knowledge of only a subset of the total nodes in a system, cooperatively route a message to the node responsible for the given key [132, 142, 207]. For example, in Chord, nodes are organized using a circular namespace and, in a naive case, can simply be routed around the circle until they arrive at the correct node [159]. To optimize this, each node maintains a "finger table" that contains increasing large steps along the circle. In this way, through a greedy algorithm, the system can resolve lookups in log time. Additionally, as with most DHT algorithms, Chord is compatible with dynamic membership; nodes are able to enter and leave at any time. Similarly, Pastry involves an overlay network based on nodeIDs, and a three tiered routing table [142]. As requests are routed through the system, each node passes it along to a node that is "closer" to the nodes responsible for housing that requested object. In this way, the authors show that message arrive at their intended destination in a logarithmic number of steps.

Information management systems attempt to provide system level awareness, while still maintaining a decentralized, distributed architecture. Some, such as SDIMS and Shruti, utilize DHT algorithms as part of their foundation [199, 200]. These information management systems aggregate information about a distributed system's state, and make it available in a way that does not collect all of the information in a central point of failure. Another approach to data aggregation is seen in Astrolabe, which eschews DHTs, in favor of a gossip-based approach [136]. Unfortunately, this approach is inefficient with certain workloads, and focuses more on data summaries that data aggregation.

3.2.2 Distributed Leadership Election

While distributed architectures offer the promise of relief from centralized points of failure, there are still a number of tasks that require a central coordinator. Thus, for tasks as varied as key distribution, routing and data aggregation, a feasible distributed system must have the ability to identify and reach consensus on a centralized leader. Of course, as with many problems within the space of distributed systems, the situation is complicated by the presence of dynamic networks in which nodes may enter and leave and any time.

The problem of leader election is traditionally, and succinctly described as having the goal of eventually electing a unique leader from a set of fixed nodes. To this end, a number of algorithms have been developed that solve this core problem within the context of different

network topographies and communications models.

One of the common approaches to leader election relies on the topology of a ring network. A global extrema, bully algorithm was devised to recover tokens in a token ring network based on each node's individual id [101]. The core of the algorithm involves each node sending its id to each of its neighbors. If a node receives an id greater than its own, it passes the id along. If a node receives its own id, by virtue of the network's logical ring structure, it has the largest id, and therefore becomes the network's leader.

The other common approach to leader election relies on diffused computation over spanning trees [45, 54, 58]. In this approach, elections begin when a potential leader begins a diffused computation. Each node propagates the election message to its neighbors, and sets the node it received the message from as its parent. An acknowledgment back to the parent node is sent only after all of a node's children have responded. Multiple simultaneous elections are often dealt with using a bully approach in which the election rooted at the highest node id wins.

Beyond the basic problem, domain specific constraints color the problem, and motivate the need for domain specific solution. For example, wireless sensor networks complicate the basic leader election problem with a physical network structure that is subject to change; nodes may be moving in and out of range [52, 178, 179]. Further, as in an evolvable system, the capabilities and health of the nodes can vary widely as nodes may exhibit different battery levels and hardware designs. To this end, a number of solutions have included modifications to the basic algorithm to take node characteristics into account in order to intelligently elect an appropriate leader.

3.3 Storage Security

Security continues to be an important driving force in storage research [139]. While numerous stopgap solutions exist, it has become generally accepted that a secure, modern system must include security as part of its fundamental design. This section presents an overview of the wide gamut of problems addressed by current research. First, I present an overview of systems that provide data secrecy, both for short and long-term data lifetimes. Following that, I discuss systems that address integrity and accountability in storage systems.

System	Secrecy	Authorization	Integrity	Blocks for Compromise	Migration
FreeNet	encryption	none	hashing	1	access based
OceanStore	encryption	signatures	versioning	m (out of n)	access based
FarSite	encryption	certificates	Merkle trees	1	continuous relocation
PAST	encryption	smart cards	immutable files	1	
Publius	encryption	password (delete)	retrieval based	m (out of n)	
SNAD / Plutus	encryption	encryption	hashing	1	
GridSharing	secret splitting		replication	1	
PASIS	secret splitting		repair agents, auditing	m (out of n)	
CleverSafe	information dispersal	unknown	hashing	m (out of n)	none
Glacier	user encryption	node auth.	signatures	n/a	
Venti	none		retrieval	n/a	
LOCKSS	none		vote based checking	n/a	site crawling
POTSHARDS	secret splitting	pluggable	algebraic signatures	$O(R^{m-1})$	

Table 3.1: Capability overview of a sampling of storage systems that enforce specific protection policies. "Blocks to compromise" lists the number of data blocks needed to brute-force recover data given advanced cryptanalysis; for POTSHARDS, we assume that an approximate pointer points to R shard identifiers. "Migration" is the mechanism for automatic replication or movement of data between nodes in the system.

3.3.1 Secrecy

The facet of security that immediately springs to mind tends to be secrecy. As Table 3.1 illustrates, many systems were designed to offer some aspect of protection. However, not all were designed for archival workloads. In this subsection, I focus on systems that aim to control who can read information. While many designs utilize the rather straight-forward application of encryption, others have opted towards unconditionally secure mechanisms such as secret splitting. None the less, I show that these systems do not fully address the specific needs of secure, long-term storage.

While a number of systems identified a need for security in order to function over public networks [86, 143], others view data security as their central goal [9, 92]. These systems, such as SNAD [112], utilize encryption as part of a robust security solution [12]. While data is encrypted on the client machine, and stored in its encrypted form, SNAD also provides enough information to authenticate both readers and writers. The authors go so far as to suggest that the storage nodes may not need to authenticate incoming requests. The computationally bound nature of cryptography makes the viability of this practice, as a long-term solution, dubious at

best; solutions that work well for shorter data-lifetimes do not always translate to long-term, archival scenarios.

Unexpected results can occur when multiple demands call for strategies that inadvertently conflict with one another. For example, a common approach to storage efficiency is the use of data deduplication. By identifying common chunks of data both within and between files and storing them only once, deduplication can yield cost savings by increasing the utility of a given amount of storage. Unfortunately, deduplication exploits identical content, while encryption attempts to make all content appear random; the same content encrypted with two different keys results in very different ciphertext. While initial work has improved the security of deduplicated storage, it still suffers from the usual problems associated with long-lived encryption [48, 161]. It remains to be seen how to combine unconditionally secure operations with data deduplication.

A number of systems attempt to offer security with reliability by encrypting data, and then generating a set of erasure coded shares from the ciphertext. Popular reliability encoding techniques include algorithms such as Reed Solomon [67], and Rabin's Information Dispersal Algorithm (IDA) [129]. These functions take data as input, and produces n shares, any m of which can be used to rebuild the input data. Compared to unconditionally secure secret splitting, IDA algorithms are less secure. However, they often incur far less storage overhead.

In e-Vault [79], files to be stored are sent to a set of archives, where integrity information is calculated over slices of data; each archive only keeps only the data slices that belong to them. These partial signatures are combined to form the full data signature. In the commercial space, CleverSafe has used this approach for secure, on-line archival data hosting using a custom six of eleven IDA algorithm [38]. Allmydata has extended this approach with the addition of Merkle trees for ensuring the integrity of data [193], a technique also used by e-Safe [8]. Evaluated as a viable long-term storage solution, all suffer from a reliance on encryption, and the problem of an inside attacker; in order to relieve the customer from having to ensure the long-term survivability of their keys, such services assume that the service provide can be trusted with users' keys.

SafeStore [85] is another example of system that utilizes information dispersal and encryption as a way of providing long-term durability. Their solution is a hierarchical, distributed system with clients at the top and Storage Server Providers (SSP) at the bottom. Data

is encoded into a number of pieces, and each of those pieces is sent to an independent SSP. There, it is further protected using another layer of encoding. The effect is both inter-SSP and intra-SSP redundancy. The second part of the SafeStore design is an auditing system based on cryptographic hashes and spot checks. The node being audited provides a report of the data that they are holding, and the auditor challenges that report by asking for random blocks that it can then hash and test against the reported manifest. There are two concerns with this approach. First, encryption occurs at the server. This implies that a user other than the data owner has access to the encryption keys. Second, providing full blocks as part of the auditing systems opens the potential for slow data leaks; even when chosen randomly, over the course of many years, auditing can reveal numerous blocks. An alternative that limits the amount of data revealed would be the use of algebraic signatures [148].

Just as with encryption, one of the direct uses of secret splitting is as a mechanism to provide data secrecy. Two systems in particular, GridSharing [169] and the Steganographic File System[13], utilize fast XOR style secret splitting. GridSharing is designed for a collaborative work environment that stresses low-latency disk access. Thus, the secret-splitting algorithms usable by GridSharing are restricted to those which can offer fast encode and decode operations. As archival storage stresses data throughput over low-latency access, a system built specifically for archival storage is able to take advantage of a wider variety of secret splitting schemes, and more complex splitting policies. The Steganographic File System (SFS) is designed to provide security as well as plausible deniability. In SFS, the disk holds a number of random blocks of data. A set of these blocks, chosen using a deterministic algorithm, is used as the random input in the basic XOR scheme. The resulting block, S' in the example illustrated in Figure 2.1, is written to disk; to an attacker, it is indistinguishable from the random blocks. The authors of SFS utilize XOR based secret splitting because the input blocks can be pre-generated; they are simply random blocks. The security of SFS is based on the fact that knowing the password and which blocks of data to combine will reconstruct data, while an adversary cannot even ascertain whether the data even exists on the system.

One of the prominent systems utilizing threshold secret splitting, PASIS [65, 197] was designed for long-term data survivability. The system consists of a number of decentralized storage nodes and a PASIS agent, which resides on the client's system. My work differs from PASIS in a number of ways. First, PASIS uses versioning as an important mechanism for

audit based security. The mostly static content of long-term archival storage largely mitigates the effectiveness of versioning strategies. Thus, my work focuses more on protection through unconditional security and noticeable access patterns. Second, the authors of PASIS specifically state that they aim to provide performance comparable to existing low-latency storage; my work relaxes this constraint, as it is designed specifically for an archival usage model. Third, PASIS utilizes a directory service to translate object requests to share requests. While this provides enhanced performance, it does provide a potential target for attack. Moreover, it could provide a malicious insider with the information needed to launch a very effective, targeted attack.

3.3.2 Integrity and Accountability

While many assume that protection implies data secrecy, LOCKSS [104, 105] is designed to guard against an adversary attempting to censor or change public documents. The purpose of LOCKSS is the long-term preservation and access of static public works such as journals and essays. The protection aspects of LOCKSS are particularly interesting. It strives to enforce a security policy that specifically does not contain a secrecy aspect but emphasizes the integrity of the system's contents. The current version of LOCKSS uses a two-level polling model with an inner-circle of more trusted peers and an outer circle of newer peers. The hosts positioned in each circle are changed through a system of churning in order to avoid dependence on any particular set of peers. Additionally, there is a system of effort-based challenges. These help guard against Sybil attacks[47], where a single entity poses as multiple systems in order to unfairly influence voting.

In addition to data secrecy, encryption has been used to enforce a variety of security policies. For example, Freenet [37] is designed for the anonymous publication of information, and uses encryption to absolve users of legal responsibility for contents stored on a their nodes. The central idea is that node owners can reasonably claim that they do not know the true contents of the encrypted data. Another important aspect of Freenet is communications anonymity. The system relies on a network in which each node only knows about their direct neighbors. Resources are located through the use of hashes, and requests are limited in the number of hops they can make. Results backtrack over the request route, copying the requested data to each node along the path. This arrangement is well suited to replicating popular data, but it does not provide for the long-term persistence of its contents.

Mnemosyne is a peer-to-peer system offering plausible deniability, and extends ideas presented in the Steganographic File System [13, 71]. Thus, the basic system involves a collection of random blocks that mask the existence of real blocks. When a new disk is added to the system, it is fully populated with random data. Real blocks consist of encrypted data, stored at an address determined through a deterministic process. Ironically, the inclusion of replicated blocks introduces the need to balance replication and the risk of overwriting a real block. As the authors themselves point out, the "birthday paradox" [40] makes this a non-trivial risk. To extend the local system into a peer-to-peer system, the block address and node address can be treated as a matrix. The authors claim that this achieves data hiding in two ways. First, data is striped across multiple nodes. Second, each node has a substrate of random blocks to hide data within.

Publius [180] is an example of a system that uses secret splitting as a key management method. A content management and publishing system, the goal of Publius is to provide both content-producer and connection anonymity [46, 133, 135] (albeit the later is through the use of third party systems). This is accomplished by replicating encrypted content and using secret splitting to distribute the key amongst a group of servers. Thus, the servers do not know what they are hosting. Users access data though special URLs, that encodes the shares location as well as a content-based name. While the system uses encryption, the worse case key-loss scenario is that content would become read only. Publius supports a unique usage model, best be described as write occasionally and read maybe, and is well suited to web based publishing. The concern, from a long-term perspective, is that write-level access is controlled through the use of cryptographic primitives. Thus, the loss of an encryption key would render data as read-only in the short term but could potentially, in time, open the system up to unauthorized content modification

In contrast to the problem of plausible deniability, an increasing demand for accountability has seen the use of encryption as an auditing tool. The authors of CATS [203] cite the "trust but verify" model as the basis of their threat model. They have created a system (for use as the building block of a larger system) that makes all players accountable for their actions. They state that three properties must be true of a correct operating system: undeniability, freshness and completeness. CATS achieves this through extensive use of asymmetric cryptography. The design proposed by Peterson et. al. [120], aims to address another aspect of accountability.

In their system, users commit to a storage state such that an audit can confirm that the data has been retained, has not been modified, and is accessible. A recent slew of legislation [2, 3] places specific demands on data retention, and has made such assurances a valuable commodity. The Peterson system is based on cryptographic hash functions and chains linking the current version of the "version authenticator" with the previous version. One issue with these systems, and a common element in systems that rely on public-key cryptography, is the implicit, trusted authority. Choosing a suitable third-party for a long-term accountability solution could be difficult; predicting the long-term stability of any given organization is difficult at best.

SLTAS, extends the idea of asymmetric encryption for digital signatures by tailoring their strategy specifically for long-term archival storage [176]. To this end, the authors' approach utilizes two important design decisions. First, an independent time-stamp authority periodically resigns the data. This occurs over all previous signatures, producing what the authors describe as an increasingly large "onion". Second, in order to guard against the eventual obsolescence of computationally bound cryptosystems, the time-stamp authority uses an increasing large key each time the onion is signed. Unfortunately, as with previous work, this approach does not guard against the catastrophic failure of cryptosystems — such as the discovery of a polynomial time prime-factorization algorithm. Additionally, it assumes that the time-stamp authority will exist for the entire lifetime of the data, and that it can properly manage and preserve the necessary keys.

Similarly, the authors of SUNDR[94] ensure the integrity of shared data on untrusted servers using a combination of content based naming, and signed, updated logging. The idea is that, if two users are able to see each other's changes, then they can detect changes that either one makes to a shared file. The authors stress a concept of fork consistency, in which the only way that one server can perform an undetected change is to fork the file and maintain a separate branch. Similar to other accountability solutions, SUNDR involves a fair amount of public-key signatures. Additionally, SUNDR does not provide data secrecy; an issue that the authors address by suggesting the use of encryption.

3.4 Cost Savings

While many systems are designed to provide new functionality, others attempt to improve on a facet of existing strategies. For example, the purpose of a number of systems

System	Media	Workload	Redundancy	Consistency	Power Aware	Comm HW
PARAID	disk	server clusters	RAID		Yes	No
Nomad FS	disk	server clusters	none		Yes	No
Google File System	disk	OLTP	replicas	relaxed	No	Yes
Copan Revolution 220A	disk	archival	RAID 5	SHA 256	Yes	Yes
Sun StorageTek SL8500	tape	backup	N+1	WORM media	No	No
RAIL	optical	backup, archival	RAID 4	opt. write verify	No	Yes
Pergamum	disk	archival	2-level R.S.	algebraic sig.	Yes	Yes

Table 3.2: Sampling of storage systems designed for economic efficiency, illustrating their diverse workloads and cost strategies.

has been lower costs, while maintaining specific service levels. As Table 3.2 illustrates, these systems have been designed for a variety of workloads, and employ different strategies in pursuit of cost savings.

Many have sought to achieve cost savings through the use of commodity hardware [59, 188]. Typically, this strategy assumes that cheaper SATA drives will fail more often than server class hardware, requiring that the solution utilize additional redundancy techniques. An example of this approach is the Google File System, which utilizes a number of storage hosts and a central master server. Thus, in GFS, files are divided into fixed sized chunks throughout the system and replicated in order to adequately deal with failures within the system. Recent studies, however, cast this assumption in a new light, showing that SATA drives often exhibit the same replacement rate as SCSI and FC disks [147].

In addition to efforts to lower storage costs through low priced media, a related effort has been spent on increasing the utility that each piece of media provides. Several systems, such as the EMC Centera system [69], Farsite [48] and the Windows Single Instance Store [23] perform deduplication on a per-file basis. Other systems, such as LBFS [113], Shark [14], and Deep Store [202], utilize a more comprehensive variable-sized chunking approach.

A number of systems equate archival storage with data backups, and favor media such as tape or optical storage over hard drives [49, 78, 172, 204]. In this strategy, removable media is utilized in an attempt to achieve cost-efficiency. Unfortunately the cost savings available with such media are often offset by the need for additional hardware (e.g. extra drive heads and robotic arms). Additionally, the random access performance of these systems is often quite

poor, which complicates distributed redundancy schemes, auditing and consistency checking. For example, Redundant Array of Inexpensive Libraries (RAIL) stores data on optical disks, and utilizes RAID 4 redundancy, but only a very high level; for every five DVD libraries, a sixth library is solely devoted to storing parity [172]. Other systems have used striped tape to increase performance [49]; later systems used extra tapes in the stripe to add parity for reliability [78].

Venti is an interesting system that combines deduplication with removable media[127]. In Venti, files are broken into fixed-sized blocks before deduplication. These blocks are coalesced into fixed-sized groups, called arenas, that are designed to facilitate the use of removable media. Unfortunately, while it presents itself as an archival system, Venti makes a few design decisions that may compromise its long-term usefulness. First, the use of removable media introduces the need to either migrate data as media evolves, or maintain aging hardware. Second, the system utilizes a centralized directory over media. While each arena includes an index over its own data, thus making the central index rebuildable, as the corpus increases this becomes increasing unfeasible.

Commodity hardware, however, is not the only avenue for realizing cost savings. An increasing amount of focus has fallen on energy efficiency as a means of cutting data center costs. One of the primary culprits of rising energy costs has been in dealing with the excess amount of heat generated by inefficient power supplies. While power supplies with an energy efficiency of 90% do exist, the typical power supply's efficiency is closer to 65-70% [174]. The resulting inefficiency usually results in excess heat, a factor contributing to high cooling costs. According to one analysis, up to 60% of energy costs are going to cool equipment [64].

The development of Massive Arrays of Idle Disks (MAIDs) generated large cost savings by leaving the majority of a system's disks spun down [39]. Interestingly, using a simulation and super computing workload, the authors found that dedicated cache disks had a very detrimental effect on the system. However, this was due to the lack of locality in the authors' evaluation workload (something that might also be common in an archival workloads).

Further work centered on the use of idle disks has expanded on the idea by incorporating strategies such as data migration, the use of drives that can spin at different speeds, spinning up subsets of disk, and power-aware redundancy techniques [121, 122, 188, 208]. While these systems realize energy savings, they are not designed specifically for archival workloads, instead attempting to provide performance comparable to "full-power" disk arrays at reduced

power. Thus, they do not consider approaches that could save even more power at the expense of high performance. For example, some MAID systems, such as those built by Copan Systems [68], use a relatively small number of server-class CPUs and controllers that can control dozens of disks. However, this approach is still relatively power-hungry because the CPU and controllers are always drawing power, reducing energy efficiency. A Copan MAID system in normal use consumes 11 W/TB [68]; this is only slightly less than the 12–15 W/TB Pergamum would require *if all disks were powered on simultaneously*, and is much higher than the 2–3 W/TB that Pergamum requires if 95% of the disks are powered off.

Ironically, the relatively recent introduction of idle disks has sparked new interest in a comparatively old idea, that of log-structured file systems (LFS) [141, 185]. The original motivation behind the development of log structured file systems was that, as caches grew larger, fewer reads would be serviced from the disk, and therefore workloads at the lowest level of the storage hierarchy would become distinctly write heavy. The key insight for the connection LFS and disk spin-down is that a system that appends writes to the end of a data log intrinsically obviates the need to try and predict accesses [56]. While the storage hierarchy may not have provided quite the cache levels expected by the authors of the original work, there may be the potential for such data structures in the write dominated workloads found in archival storage.

The StorageTek 5800 is a Sun Microsystems product derived from the HoneyComb project [171]. It is designed for unstructured, static data (archival). The system is fully distributed as requests for data can be directed to any node in the system. Each node is a self-contained unit including a high-power processor, multiple drives, multiple Ethernet ports and a power supply. Cost savings come from the use of commodity hardware such as SATA hard drives, low licensing costs, and low management overhead. For reliability, they utilize Reed Solomon encoding in 5+2 arrangement. While designed for archival workloads, the system does not take power in consideration. The processors they use are very high power, and the placement algorithm does not take into account the number of disks that must be spun up. In some cases, they cite that sixty disks will be involved in a single write. Lastly, while they discuss scrubbing, they do not indicate at what frequency this occurs [149].

HYDRAstor is another system developed specifically for secondary storage[50]. The system consists of a number of back-end storage appliances, and a front-end of access nodes. In order to achieve high storage efficiency, HYDRAstor stores files as a series of immutable,

variable-sized blocks. Content based addressing provides for a level of deduplication over blocks. Cost savings come from the use of commodity hardware, however the selection of highly reliable server-grade equipment could incur a heavy price tag. Further, the system is not designed for low energy utilization, and its hash based placement algorithm for chunks conspires against disk spin-down strategies.

IBM's brick storage project, IceCube, uses federated groups of intelligent storage devices to provide low-latency, reliable storage [192]. Each brick holds on the order of tens of small drives and a relatively high power desktop processor. Currently each brick supplies on the order of a single terabyte, but requires 200W. Their system is designed for front line storage and allows administrators the flexibility to choose their own reliability versus cost trade offs. One of the interesting notes is that they believe that bricks should minimize internal redundancy in order to lower costs. The primary cost savings in IceCube comes from the system's strategy of deferring management to a convenient time, as opposed to a more ambitious goal of self-management. To this end, the system still includes centralized administrator nodes and requires administrative input. Surprisingly, despite the system's use of intelligent storage devices, IceCube rejects the notion of evolvability in several ways. First, the current version relies on highly specialized interconnects and water-cooled racks. Second, the three dimensional structure of the system prevents failed, interior nodes from being removed. This choice wastes floor space, and suggests that a fork-lift upgrade is inevitable.

Another brick storage project, BitVault, follows the model of utilizing numerous, intelligent storage devices [205, 206]. Like Pergamum, the authors of BitVault identify archival storage's need for cost efficient storage. Unfortunately, there are three issues that prevent their solution from maximizing the cost saving potential of disk based archiving. First, BitVault does not address power efficiency (although it is identified as an area of future work). Second, BitVault achieves reliability through the use of file replication. While the advantage of this design is that it facilitates the use of per file replication choices, it adversely affects the storage efficiency of the system. Third, BitVault utilizes a global knowledge approach to routing, which further lower storage efficiency and reduces its scalability.

An active area of research, storage management tools are designed to provide administrators with the knowledge they need to make good decisions [82, 183]. Hippodrome takes a very proactive role in storage management, as the authors express the belief that administration

is quickly becoming one of the dominant costs in modern storage systems [11]. The core of the authors' algorithm involves an iterative process made up of four key components. First there is the analyzer that is used to obtain a custom workload. Then there is a performance modeler and solver that is used to produce a working and valid storage layout model. Finally there is an implementor that is used to migrate the system to the new model. There are two issues that I feel the authors did not address in their implementation. The first issue that I feel was omitted was the safety of their migration technique. One might imagine that any migration would come with an inherent risk. The second issue, which is related to the first, is the question of simulation. The field of optimization theory is an area rich with techniques and rigorous models for modeling and optimizing systems. The authors fail to mention if any of these models might be useful.

Similarly, the authors of Zodiac focus on policy-aware impact analysis [154]. Central to its design is a session based component that allows administrators to pose queries to the system. Being a session based model, the queries have the flexibility of being incremental in nature. This is in contrast to SQL type queries in which each query is separate and atomic. Further, a policy classification mechanism assists in gathering relevant SAN data in order to reduce the solution space for a number of key optimizations. For example, Zodiac includes a caching mechanism in which SAN metadata is distributed throughout the system thus providing faster access. The authors state that this improves impact analysis due to the commonality of data accessed. Additionally, Zodiac includes a system of aggregation, which helps improve the performance of certain queries. All of this is used by an evaluation and visualization components that provides the administrator with visual feedback over the SAN structure that was input to the system.

As part of an eventual goal of fully automatic system administration, Self-* has focused on performance tuning, providing administrators and tuning agents with the ability to model changes to basic tuning parameters [5, 57]. The authors make the claim that such management abilities must exist in systems as a fundamental feature, drawing an analogy to early attempts at security that amounted to add-ons to existing systems. Their solution consists of a management layer, above the storage layer, populated with automation agents and administrative interfaces. A key point that their early work identifies is the need to track and log information per request. In this manner, logged data provides a complete picture of the re-

quest’s life cycle; isolated information makes it difficult to identify bottlenecks and correlated events.

Chapter 4

POTSHARDS, Long-Term Archival Security

> One never reaches home, but wherever friendly paths intersect the whole world looks like home for a time.
>
> Hermann Hesse

Few would disagree that there has been a tremendous shift towards writing business data and our personal histories as digital data. In the professional sector, the demand for data security is not surprising. However, even for personal artifacts, there is a distinct need for long-term data protection. For individuals, archival storage is being called upon to preserve sentimental and historical data such as photos, movies and personal documents. This information often needs to be stored securely [131]; data such as medical records and legal documents that could be important to future generations must be kept indefinitely but must not be publicly accessible.

While storage security is a relatively mature area of research covering a large corpus of work, the long lifetimes of archival storage demand a thorough reexamination of a number of fundamental assumptions. With POTSHARDS, I demonstrate my thesis statement — archival storage is a first class storage category that requires solutions tailored for long-lived data — by describing an approach to security designed specifically to meet the needs of long term data storage.

The goal of a secure, long-term archive is, therefore, to provide security for relatively static data with an indefinite lifetime. More specifically a secure archive seeks to provide three, long-term features: secrecy, recoverability and integrity. The first, long-term secrecy,

aims to ensure that the data stored must only be viewable by authorized readers. The second, recoverability, is akin to availability and stipulates that data must be available and accessible to authorized users within a reasonable amount of time, even to those who might lack a specific key. The third, integrity, ensures that the data read is the same as the data written.

The privacy aspect of POTSHARDS is achieved through unconditionally secure mechanisms, increased attack survivability and malicious activity detection. Secret splitting provides attack resilience because, unlike pure encryption, it is unconditionally secure and requires the adversary to collect multiple pieces of data to reconstitute *any* portion of the original block. Further, the likelihood of detecting malicious data access is probabilistically increased through a sparse namespace; requests for shares that do not exist are easy to detect. Compounding the attack detection's effectiveness, an attacker that attempts to use the approximate pointer to make a targeted attack would need to steal every share in the indicated region along with every share in the region indicated by those shares and so forth.

The recovery and availability strategy of POTSHARDS enables the reconstruction of data from the secret shares alone. Thus, even with no outside index to connect data blocks and secret shares, a user's data can be recovered. This is especially important in long-term archival scenarios in which data may have a potentially indefinite lifetime [19, 162]. My approach is based upon the use of approximate pointers, which provide clues about inter-share relationships. These clues supply enough information to allow recovery but require a lot of shares, a necessity that is difficult for an adversary to meet.

4.1 POTSHARDS Overview

Since POTSHARDS was designed specifically for secure, long-term storage, I identified three basic design tenets to help focus my efforts. First, I assume that encrypted data can be read by anyone given sufficient CPU cycles and advances in cryptanalysis. Put another way, if an attacker obtains encrypted data, the plaintext will eventually be revealed. Second, for long-term survivability, data must be recoverable without any information from outside the set of archives; fulfilling requests in a reasonable time cannot require any outside data, such as external indexes or encryption keys. Third, I assume that individuals are more likely to be malicious than an aggregate. Thus, the system trusts groups of archives, even though it does not trust individual archives. The chance of every archive in the system colluding maliciously

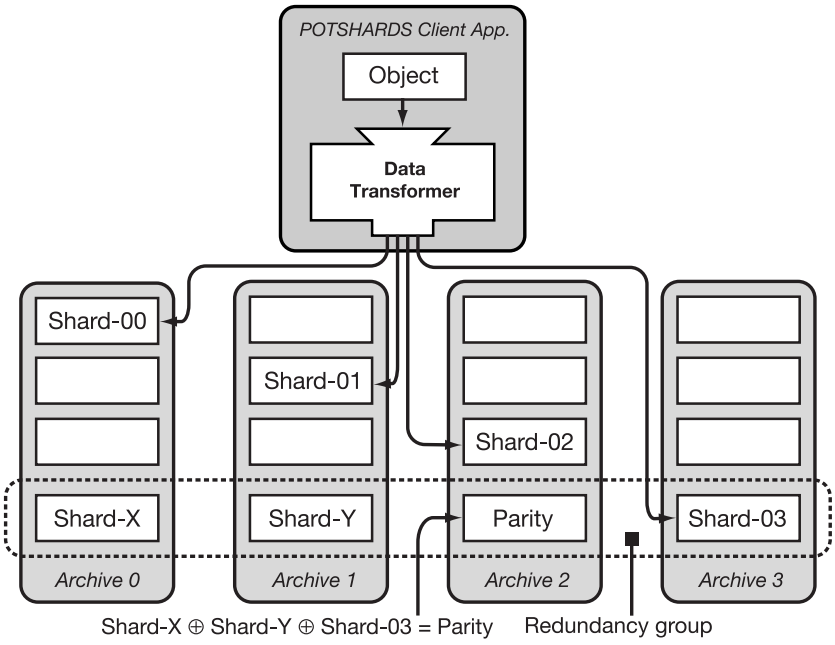


Figure 4.1: An overview of POTSHARDS showing the data transformation component of the client application producing shards from objects, and distributing them to independent archives. The archives utilize distributed RAID algorithms to securely recover shards if an archive is lost.

is small; the system allows rebuilding of stored data if all archives cooperate.

POTSHARDS is structured as a client application communicating with a number of independent archives. Though the archives are independent, they assist each other through distributed RAID techniques to protect data against archive loss. Users store their data within the system using a POTSHARDS client, which splits their data it into secure *shards*. These shards are then distributed to a number of archives, where each archive exists within its own security domain. The read procedure is similar, but reversed; a user utilizes the POTSHARDS client to request shards from archives and reconstitute the data.

Users access the system through a POTSHARDS client, which has three primary functions. First, the client handles all *data transformation* duties. For writes, as shown in Figure 4.1, this involves generating *shards* from *objects* through the use of secret splitting techniques. For reads, the process is reversed, shard identifiers are used to fetch shards from the archives, and objects are reconstructed. Second, the client is responsible for distributing shards to archives such that no single archive has enough shards to reconstruct data. Third, as the client resides on a system separate from the shards, the POTSHARDS client is responsible for han-

dling communication between the user and the archives. The advantage of this arrangement is that data never reaches an archive in unsecured form, and multiple CPU-bound data transformation processes can generate shards in parallel for a single set of physical archives. Of course, as in any security application, careless implementation of the POTSHARDS client can introduce unforeseen compromises; adversaries can take advantage of carelessly cached passwords and other such key material.

Shards are stored in a series of independent archives, that function similar to financial banks; they are relatively stable and they have an incentive (financial or otherwise) to monitor and maintain the security of their contents. While security is strengthened by distributing shards amongst the archives, it is important that each archive can demonstrate an ability to protect its data. Other benefits of archive independence include reducing the effectiveness of insider attacks and making it easier to exploit the benefits of geographic diversity. For these reasons, even a single entity, such as a multinational company, should still maintain multiple independent archives.

In order to limit the effectiveness of insider attacks, there is no central index over shards. Rather, users maintain a private index that maps their data to shards. This is made possible by the fact that POTSHARDS enables the reconstruction of data from the shards alone. This private index, which could be contained on a physical token such as a smart-card, allows normal read operations to take place quickly because the user would know exactly which shards to request and how to combine them. If, however, a user loses their index, or never had one, it can be regenerated in a reasonable amount of time. By removing the need for an omniscient, central authority, the risk of a malicious insider is mitigated.

4.1.1 Security Techniques

Security in POTSHARDS is provided by two mechanisms: a sparsely populated, global namespace, and unconditionally secure secret splitting. With secret splitting, an intruder must collect multiple shards in order to read any data, and the sparse namespace makes attacks more noticeable by increasing the chances that an intruder will request shards that do not exist.

Secret splitting provides the secrecy in POTSHARDS with a degree of future-proofing —it can be proven that an adversary with infinite computational power cannot gain any of the original data, even if an entire archive is compromised. Further, these algorithms provide

file secrecy without the need for the key and algorithm rotations that traditional encryption introduces; perfect secret splitting is unconditionally secure. Thus, POTSHARDS is not forced into maintaining complex key histories.

A number of secret splitting algorithms, known as threshold schemes, produce a set of n shares, any $m < n$ of which are needed to rebuild the original data. While POTSHARDS can utilize such schemes, it does not rely on them for the system's reliability. Rather, the small amount of redundancy these algorithms offer allows POTSHARDS to handle transient archive unavailability by not requiring that a reader obtain *all* of the shards for an object.

In addition to uniquely identifying data entities in POTSHARDS and improving attack detection, the global namespace enables the use of secret splitting algorithms by imposing an ordering over entities. Many threshold schemes, such as those that rely on linear interpolation [153], require both the shares and a specific ordering of those shares for reconstruction. Preserving the ordering over a tuple of shards is easily accomplished by naming the shards in ascending order, according to their location within the full shard tuple. In this way, names impose a total ordering over a complete tuple of shards.

4.1.2 Reliability and Availability Techniques

POTSHARDS provides reliability and availability through two distinct recovery strategies. First, as Figure 4.2 illustrates, the shards that reconstruct a data block form a circularly-linked list, allowing a specific user's data to be recovered from their shards alone. This ring of shards is generated within the transformation components, as part of the ingestion process. Second, the loss of an entire archive is handled using distributed RAID techniques, across multiple independent archives. This two level approach allows POTSHARDS to scale the recovery to the size of the data loss.

In the absence of the index over a user's shards, approximate pointers can be used to recover data from the shards alone. Such a scenario could occur if a user loses the index over their shards, or in a long-term time-capsule scenario in which a future user may be able to access the shards that they have a legal right to, but have no idea how to combine them.

Approximate pointers enable the use of secret splitting by providing a built-in method of "key recovery"; knowing which secret shares to combine is analogous to an encryption key because it is the secret that transforms ciphertext into plaintext. Without the clues provided by

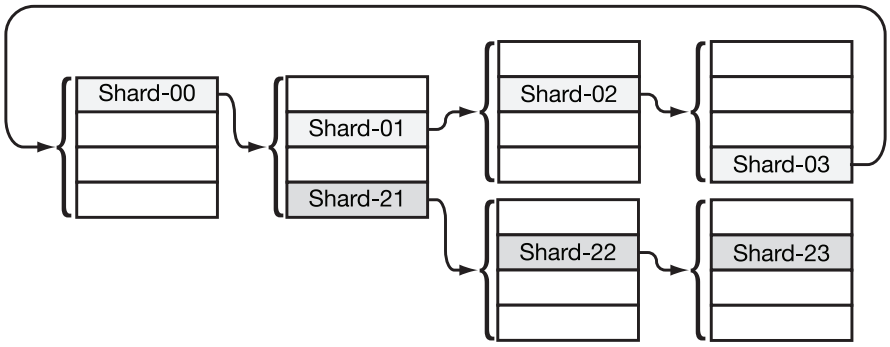


Figure 4.2: Approximate pointers point to R "candidate" shards ($R = 4$ in this example) that might be next in a valid shard tuple. Shards_{0X} make up a valid shard tuple. If an intruder mistakenly picks shard_{21}, he will not discover his error until he has retrieved sufficient shards and validation fails on the reassembled data.

approximate pointers, recovery involves testing every possible combination of shards, making it an intractable problem. In contrast, while direct pointers would make recovery trivial, it would also compromise security; an adversary with one shard could easily make targeted attacks for the rest of the shards. Thus, the advantage of approximate pointers is that, by indicating a region and utilizing namespace sparseness, targeted attacks are much more difficult, and brute force attacks would be quite noticeable. Thus, secrecy is not unduly affected, providing a worthwhile tradeoff for slower recovery times if a block's shard list is lost.

To deal with larger scale losses, the archive layer in POTSHARDS consists of independent archives utilizing secure, distributed RAID techniques. As Figure 4.1 shows, archive-level redundancy is computed across sets of *unrelated* shards, so redundancy groups provide no insight into shard reassembly. POTSHARDS includes two novel modifications beyond the distributed redundancy explored earlier [29, 160]. The first is a secure reconstruction procedure, described in Section 4.2.3.1, that allows a failed archive's data to be regenerated in a manner that prevents archives from obtaining additional shards during the reconstruction; shards from the failed archive are rebuilt only at the new archive that is replacing it. Second, POTSHARDS uses algebraic signatures [148] to ensure intra-archive integrity as well as inter-archive integrity. Algebraic signatures have the desirable property that the parity of a signature is the same as the signature of the parity.

4.2 Implementation Details

This section details the components and security model of POTSHARDS, and how each contributes to providing long-term, secure storage. First, I describe how objects in POTSHARDS are named, and present two naming dangers that, if ignored, could compromise data security. Second, I describe the POTSHARDS client in detail, including how it produces shards from objects. Third, I describe the role of the archives, including how they securely rebuild data if an archive is lost. Fourth, I describe the role and construction of the user's private index. Finally, I describe how approximate pointers are used to recover a user's data from the shards alone.

4.2.1 Naming

All of the data entities in POTSHARDS, both higher level entities such as objects, as well as lower-level secure entities such as shards, exist within a single 128 bit namespace. Each identifier contains two portions. The first 40 bits of the name identify the user in the same manner as a bank account is identified by an account number. The remaining 88 bits are used identify the data entity.

While names for high-level POTSHARDS entities, such as objects, can be generated fairly easily, the names of lower-level entities, such as shards, must be chosen more carefully; shard names and approximate pointer rings directly affect security and recovery. Two naming and ring formation scenarios in particular have the potential to compromise security. First, a poorly chosen ring of shards could inadvertently reduce the search space of a targeted attack. Second, poorly named shards could leave the potential namespace fan-out under-utilized.

Careless naming and ring formation can inadvertently provide an attacker with information that effectively reduces the search space for the next shard. For example, if the shards in a tuple are ordered $S_1, S_2, \ldots, S_n$ and shard S_i always points to shard S_{i+1}, an attacker would know that the name of the next shard must be greater than the current shard. Now suppose that shard S_i itself is within the range indicated by the approximate pointer to S_{i+1}. As illustrated in Figure 4.3, the attacker would know that $S_i < S_{i+1}$, and thus can narrow down the search space.

To avoid revealing information through shard names, a simple randomizing procedure can be used to permute the total ordering of the shard tuple into a separate ring order. This procedure proceeds as follows:

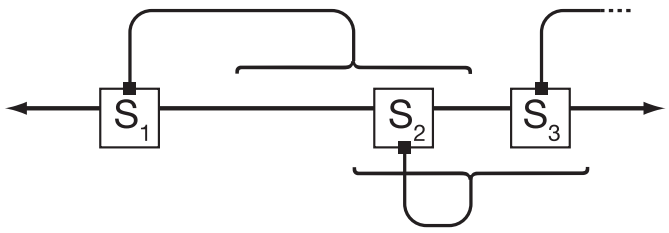


Figure 4.3: Example of a situation in which careless naming has reduced the search space indicated by an approximate pointer. If shard ordering is not randomized, an adversary would know that S_3 must be greater than S_2 and thus would only need to search the region above S_2.

1. Determine the names that will be used for the shards (e.g.: (S_0, S_1, S_2, S_3))
2. Create the shards and name them in ascending order so that their position within the tuple is preserved by the total ordering imposed by their names
3. Randomize the order of these shards (e.g.: (S_2, S_1, S_0, S_3))
4. Use approximate pointers to form a ring based on this randomized order. Thus, the next shard can exist in any portion of the namespace, regardless of the current shard's name.

Another danger involves the under-utilization of the fan-out that can be achieved with approximate pointers. Since approximate pointers indicate a region, as opposed to a single address, they have the potential to greatly increase an adversary's workload. An ideal arrangement is achieved if each shard in a given region points to a different region. In this scenario, the adversary would need to acquire each shard in each of those diverse regions. Figure 4.4 illustrates an example in which the shard names and approximate pointers are configured poorly, resulting in little fan-out. The effect is a greatly reduced workload for the adversary— the attacker would only need to acquire the shards of overlapping regions once, rather than having to steal a given shard once for each predecessor that could point to it.

In order to ensure the greatest fan-out, careful shard naming and linking is required. Since users maintain an index of object to shard mappings, naming can proceed with knowledge of previously named shards. An area of future work could be to further develop intelligent naming techniques; the security of the system is greatly influenced by the namespace and the links between shards, making this a particularly important area to examine.

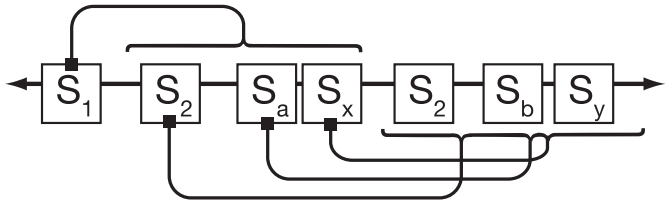


Figure 4.4: Example of a situation in which careless naming has underutilized the potential of approximate pointers to increase the fan-out of linked shards. Ideally, S_2, S_a, and S_x would all point to different regions.

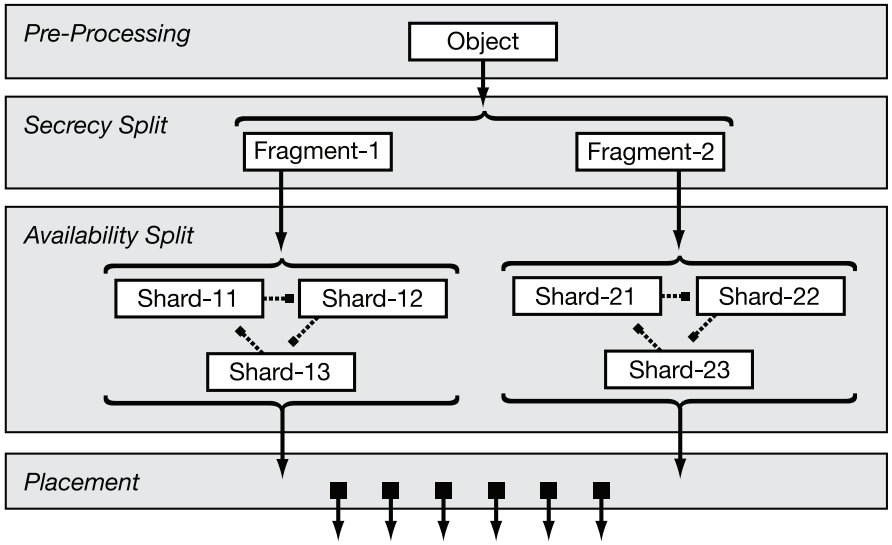
4.2.2 POTSHARDS Client: Data Transformation

One of the primary tasks of the POTSHARDS client is to perform the data transformation that produces shards from user data. As Figure 4.5 illustrates, the client is composed of four layers, and utilizes three unique data entities. During the ingestion of data, the pre-processing layer is responsible for producing fixed-size *objects* from user files. Objects are then transformed into *fragments* in a secret split tuned for secrecy. A second split occurs, this time tuned for availability, which transforms fragments into *shards*. Finally, the placement layer is responsible for distributing a set of shards to the archives. Extraction is similar, but reversed; shards are requested from the archive, combined into fragments, and those fragments are combined into objects.

The two levels of secret splitting provide three important security advantages. First, as Figure 4.5 illustrates, two levels of splitting results in a tree, providing extra security through increased fan out; even with all of the members of a shard tuple, an attacker can only rebuild a fragment, which provides no information about the shards for the other fragments. Second, as secret splitting algorithms present varied features, each split can be independently tuned for a specific property, and can select the algorithm best suited to that property. Third, it enables recovery by allowing useful metadata to be stored with the fragments; this data will be kept secret by the second level of splitting.

4.2.2.1 Pre-Processing Layer

When a user submits data to the POTSHARDS client for ingestion, objects are created from the user's files in a three step process. First, each file is divided into a series of fixed-sized blocks. As the system is designed for archival workloads, these blocks are on the order of several



(a) Four data transformation layers in POTSHARDS.

Module	*Input*	*Output*
Pre-processing	file	object
Secrecy split	object	set of fragments
Availability split	fragment	set of shards
Placement	set of shards	msgs for archives

(b) Inputs and outputs for each transformation layer.

Figure 4.5: The transformation component consists of four levels. Approximate pointers are utilized at the second secret split. Note that locating one shard tuple provides no information about locating the shards from other tuples.

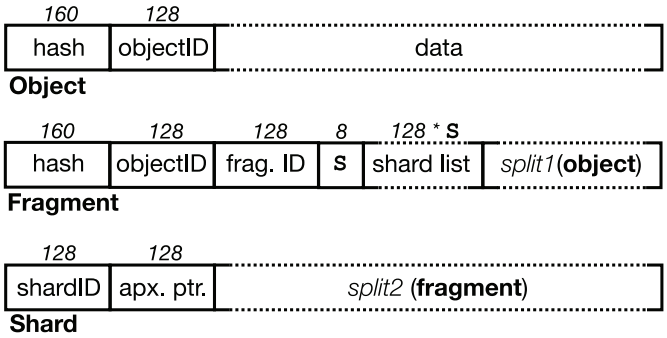


Figure 4.6: Data entities in POTSHARDS, with size (in bits) indicated above each field. Note that entities are not shown to scale relative to one another. S is the number of shards that the fragment produces. split1 is an XOR secret split and split2 is a Shamir secret split in POTSHARDS.

hundred kilobytes to a megabyte in size. Second, as Figure 4.6 details, an object identifier is generated and appended to the block. Third, a hash over the block and id is generated and appended. This hash is used to confirm a successful rebuild during reads. It does not, however, compromise security as it included in the unconditionally secure secret split in the later stages of shard production.

4.2.2.2 Secret Splitting Layers

Fragments are generated from objects at the first of two secret splits that occurs in the secret splitting layers. This first split is tuned for secrecy, and currently uses an XOR-based algorithm that produces *n* fragments from an object, all *n* of which are required for reconstruction. To ensure security, the random data required for XOR splitting can be obtained through a physical process such as radio-active decay or thermal noise.

As Figure 4.6 illustrates, each fragment contains metadata that assists in reconstruction and recovery. First, as in the object, a hash over the entire fragment serves to confirm a successful reconstruction. Second, the object identifier that this fragment contributes to aids in reconstruction; if a user is able to reproduce all of their fragments, this identifier assists in combining them into objects. This approach does not compromise security, as reconstructing a single fragment provides no information about which shards form the other fragments for a given object. Third, the fragment contains its own id. Finally, each fragment contains a list of the shards it produces.

A tuple of shards is produced from a fragment using another layer of secret splitting. This second split is tuned for availability, and therefore the current implementation of POTSHARDS uses an *m* of *n* secret splitting algorithm [153]. As mentioned in Subsection 4.1.1, this allows reconstitution in the event that an archive is down or unavailable when a request is made.

As Figure 4.6 shows, shards contain no information about the fragments that they make up. They do, however, include two pieces of metadata. First, they include their own shard id. Second, they include an approximate pointer to a random shard from the same shard tuple, as described in Subsection 4.2.1.

The approximate pointers can be implemented using one of two approaches. First, the *bitmask method*, indicates a region, R, by masking off the low-order r bits ($R = 2^r$) of an actual address, hiding the true value. The drawback of the bitmask method is the coarse level of granularity that can be achieved. It does, however, have the advantage that the size of the region indicated by the approximate pointer is relatively self-evident: it is straightforward to see how many bits are masked off (set to zero) in an address. Second, the *range method* randomly selects a value within $R/2$ above or below the actual address. In contrast to the bitmask method, the granularity offered by the range method is quite good. However, it is not self-evident from the approximate pointer how large the range is. Our implementation uses the latter approach.

One drawback of the two-level secret splitting approach is the resulting increase in storage requirements. A two-way XOR split followed by a 2 of 3 secret split increases storage requirements by a factor of six; distributed RAID, and metadata further increases the overhead. If a user desires to offset this cost, data can be submitted in a compressed archival form [202]; compressed data is handled just like any other type of data.

4.2.2.3 Placement Layer

During ingestion, the placement layer is responsible for mapping shards to archives. The decision takes into account which shards belong in the same tuple and ensures that no single archive is given enough shards to recover data. During extraction, the placement layer is responsible for requesting shards from archives.

This layer contributes to security in four ways. First, since it is part of the data transformation component, no knowledge of which shards belong to an object need exist outside of

the client. Second, the effectiveness of an insider attack at the archives is reduced because no single archive contains enough shards to reconstitute any data. Third, the effectiveness of an external attack is decreased because shards are distributed to multiple archives, each of which can exist in their own security domain. Fourth, the placement layer can take into account the geographic location of archives in order to maximize the availability of data.

4.2.3 Archive Design

Persistent storage of shards is handled by a set of independent archives that actively monitor their own security, and question the security of the other archives. The archives do not, however, know which shards combine to form a fragment, or which shards contribute to a given object. Thus, a compromised archive does not provide an adversary with enough shards to rebuild user data. Additionally, it does not provide an adversary with enough information to launch a targeted attack at the other archives. Absent such precautions, the archive model would likely weaken the strong security properties provided by the other system components.

Since POTSHARDS is designed for long-term storage, it is inevitable that disasters will occur, and archive membership will change over time. To deal with the threat of data loss from these events, POTSHARDS utilizes distributed RAID techniques. The space at each archive is divided into fixed-sized blocks, each holds either shards or redundancy data. Archives then agree on distributed, RAID-based methods over these blocks.

As in other distributed RAID systems [29, 160], fault-tolerant, distributed storage is achieved by computing parity across unrelated data in wide area redundancy groups. Given an (n,k) erasure code, a redundancy group is an ordered set of k data blocks and $n-k$ parity blocks where each block resides on one of n distinct archives. The redundancy group can survive the loss of up to $n-k$ archives with no data loss. The current implementation of POTSHARDS has the ability to use Reed-Solomon codes or single parity to provide flexible and space-efficient redundancy across the archives.

When shards arrive at an archive for storage, ingestion occurs in three steps. First, a random block is chosen as the storage location of the shard. Second, the shard is placed in the last available slot in that block. Third, the corresponding parity updates are sent to the proper archives. The failure of any parity update will result in a roll-back of the parity updates, and re-placement of the shard into another block.

An integral part of preserving data, POTSHARDS actively verifies the integrity of data using two different forms of checking. First, each archive actively monitors the integrity of its own contents using stored hashes. Second, inter-archive integrity checking is performed using algebraic signatures [148] across the redundancy groups. Algebraic signatures have the property that the signatures of the parity equals the parity of the signatures. This property is used to verify that the archives in a given redundancy group are properly storing data and are performing the required internal checks.

Secure, inter-archive integrity checking is achieved through algebraic signature requests over a specific interval of data. A check begins when an archive asks the members of a redundancy group for an algebraic signature over a specified interval of data. The algebraic signature forms a codeword in the erasure code used by the redundancy group, and integrity over the interval of data is checked by comparing the parity of the data signatures to the signature of the parity. If the comparison check fails, then the archive(s) in violation may be found as long as the number of incorrect signatures is within the error-correction capability of the code. This approach is efficient and secure as signatures are typically only a few bytes, and only leak b bytes for signatures of length b.

4.2.3.1 Secure Archive Reconstruction

Reconstruction of data can pose a significant security risk because it involves many archives and considerable amounts of data passing between them. POTSHARDS mitigates this risk through a secure protocol that allows each archive to assist in the reconstruction of failed data, without revealing any information about its data. Further, the reconstruction procedure is performed in multiple rounds in order to prevent collusion between archives.

The recovery protocol begins with the confirmation of a partial or whole archive failure and, since each archive is a member of one or more redundancy groups, proceeds one redundancy group at a time. If a failure is confirmed, the archives in the system must agree on the destination of recovered data. This fail-over archive is chosen based on two criteria. First, the fail-over archive must not be a member of the redundancy group being recovered. Second, the fail-over archive must have the capacity to store the recovered data. Due to these constraints, multiple fail-over archives may be needed to perform reconstruction and redistribution. Future work will include ensuring that the choice of fail-over archives prevents any archive from

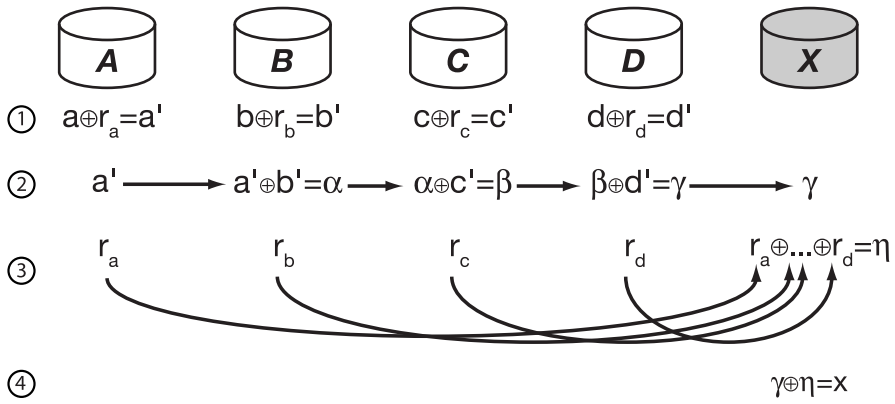


Figure 4.7: A single round of archive recovery in a RAID 5 redundancy group. Each round consists of multiple steps. Archive N contains data n and generates random blocks r_n.

acquiring enough shards to reconstruct user data.

Once the fail-over archive is selected, recovery occurs in multiple rounds. A single round of the secure recovery protocol is illustrated in Figure 4.7. In this example, the available members of a redundancy group collaborate to reconstruct the data from a failed archive onto a chosen archive, X. An archive, which cannot be the fail-over, is appointed to manage each round (in Figure 4.7, archive A has been selected). The managing archives determines the ordering for the round and generates a request containing an ordered list of archives, the id of the block to regenerate, and a data buffer. Each archives in the list then proceeds as follows:

1. Request α involving local block n arrives at archive N.
2. The archive creates a random block r_n and computes $n \oplus r_n = n'$.
3. The archive computes $\beta = \alpha \oplus n'$ and removes its entry from the request
4. The archive sends r_n directly to archive X.
5. β is sent to the next archive in the list.

This continues at each archive until the chain ends at archive X and the block is reconstructed. The commutativity the rebuild process allows decreases the likelihood of data exposure by permuting the order of the chain in each round. This procedure is easily parallelized and continues until all of the failed blocks for the redundancy group are reconstructed. This

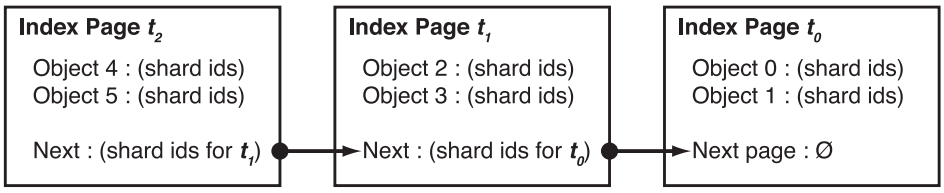


Figure 4.8: User index stored in POTSHARDS as multiple pages. The initial page was created at time t_0, subsequent pages at times t_1 and t_2 respectively. By knowing just the shards to the newest page, the user can extract the entire index.

approach can be generalized to any linear erasure code; as long as the generator matrix for the code is known, the protocol remains unchanged.

4.2.4 User Indexes

While approximate pointers join the shards within the systems, the *exact* names are returned to the user during ingestion, along with the archive placement locations. Typically, a user maintains this information and the relationship between shards, fragments, objects, and files in an index to allow for fast retrieval. In the general case, the user consults her index and requests specific shards from the system. This index can, in turn, be stored within POTSHARDS, resulting in an index that can be rebuilt from a user's shards with no outside information.

It is important to note that, while the index does contain the information describing which shards correspond to fragments and objects, it does not provide the information needed to obtain those shards. An attacker with a user's index will still need the information needed to authenticate to the archives containing the user's shards. Of course, as with any security scheme, an adversary with enough information — in the case of POTSHARDS, the user's index and enough authentication information to sufficiently pose as the user — is assumed to have acquired full access to the user's data.

The index for each user can be stored in POTSHARDS as a linked list of index pages, with new pages inserted at the head of the list, as shown in Figure 4.8. Since the index pages are designed to be stored within POTSHARDS, each page is immutable. When a user submits a file to the system, a list of mappings from the file to its shards is returned. This data is recorded in a new index page, along with a list of shards corresponding to the previous head of the index list. This new page is then submitted to the system and the shard list returned is maintained as the

new head of the index list. These index root-shards can be maintained by the client application or even on a physical token, such as a flash drive or smart card.

The approach of private, per user indices has a number of advantages compared to a single, centralized index. First, since each user maintains his own index, the compromise of a user index does not affect the security of other users' data. Second, the index for one user can be recovered with no effect on other users. Third, the system does not know about the relationship between a user's shards and their data.

While the index over a user's shard contains the information needed to rebuild a user's data, it differs from an encryption key in two important ways. First, unlike an encryption key, the user's index is not a single point of failure. If the index is lost or damaged, it can be recovered from the data without any input from the owner of the index. Second, full archive collusion can rebuild the index. If a user can prove a legal right to data, such as by a court subpoena, than the archives can provide all of the user's shards and allow the reconstitution of the data. If the data was encrypted, the files without the encryption key might not be accessible in a reasonable period of time.

4.2.5 Recovery with Approximate Pointers

Recovery through the use of approximate pointers is based upon the graph structure that approximate pointers impose over a set of shards. Each shard is a named vertex in the graph, with an edge between it and every other vertex within the region defined by the shard's approximate pointer. The relationships described by this graph are used to recover data through the use of two recovery algorithms: the *naïve* approach, and the more efficient *ring heuristic*. Both approaches are based on knowing the spitting parameters, m of n, that produced the shards.

With both reconstruction strategies, the process starts the same way. Once a user determines that she must recover her data, perhaps due to a lost index, she begins by collecting her shards. As subsection 4.2.1 described, the user's shards can be identified by the initial, user id portion of the shard name. The operation to collect all of the shards could differ for each archive. Additionally, releasing all of a user's shards is a potentially dangerous; a lot of data could be compromised. Therefore, this operation should require a higher level of authorization and clearance.

In the first recovery strategy, the naïve approach, the solution-space is reduced by

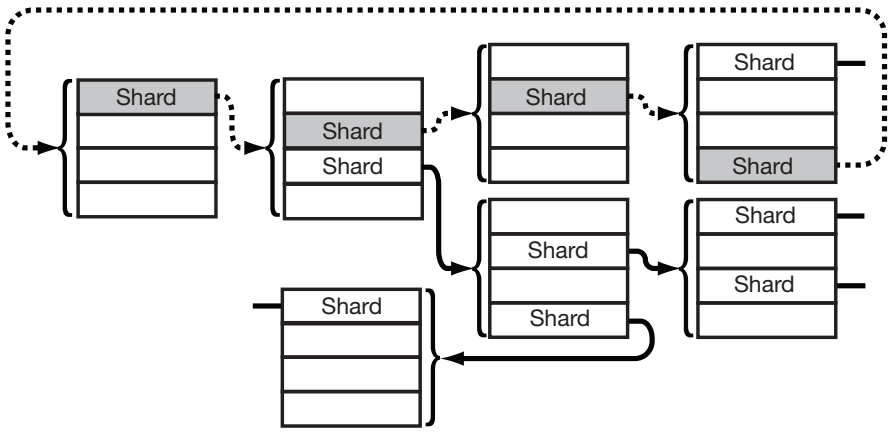


Figure 4.9: Recovery example where each approximate pointer indicates a region of four shard names. If shards are produced using a 2 of 4 split, the ring heuristic reveals one recovery candidate based on its circular linked list structure of *exactly n*, four, shards (shaded shards and dotted approximate pointers). In contrast, the naïve approach of testing paths of length *m*, 2, would result in many more potential recovery candidates.

limiting reconstruction attempts to paths of length *m*. This approach can be useful when less than the full set of shards are available; with less than a full set of shards, the user may not have all *n* shards that reconstruct a fragment. Unfortunately, a number of factors conspire to make this approach less than ideal. First, as Figure 4.9 illustrates, while still better than a purely brute force based approach, there are still a fair number of paths of length *m*, and therefore many possible candidates for reconstruction. Second, a side effect of the randomization discussed in Subsection 4.2.1, is that reconstruction with less than a complete tuple of shards is time consuming; secret splitting is expensive, and a user with less than *n* shards does not have a total ordering, and must attempt recovery on multiple permutations. For example, suppose that a user posses a chain of three of the five shards, S_a, S_b, S_c, resulting from a 3 of 5 threshold split. If the inter-shards links were not formed using the randomization method, but rather were simply formed using the name order, reconstruction would potentially involve testing three shard tuples: $\langle S_a, S_b, S_c, \emptyset, \emptyset \rangle$, $\langle \emptyset, S_a, S_b, S_c, \emptyset \rangle$ and $\langle \emptyset, \emptyset, S_a, S_b, S_c \rangle$. However, if the shards were connected using the randomized method, reconstruction attempts would need to include combinations with interspersed empty shards, such as $\langle \emptyset, \emptyset, S_a, S_b, S_c \rangle$, $\langle \emptyset, S_a, \emptyset, S_b, S_c \rangle$, $\langle \emptyset, S_a, S_b, \emptyset, S_c \rangle$, ...

The second strategy, the ring heuristic, utilizes the circular structure of the shard

tuples, depicted in Figure 4.9. This approach only attempts to reconstruct cycles of length n, and provides two important advantages. First, it efficiently reduces the potential solution space to a manageable number of recovery candidates. Second, because the ring heuristic identifies all n members of a shard tuple, the shard names impose a full ordering. One disadvantage of this approach is that it requires a lot of shards. However, given an incomplete set of shards, the ring heuristic can be used as a first pass algorithm to quickly recover the full shard tuples and reduce the solution space for the remaining, unrecovered secrets. These can then be recovered using the naïve approach.

In addition to approximate pointers, there are other hints in the structure of the data entities, illustrated in Figure 4.6, that are useful with both the naïve approach and the ring heuristic. First, a hash of the fragment is used to confirm a successful reconstruction. Second, each reconstructed fragment includes a list of the secret shards that it produces. Using this list, reconstruction of a secret from less than n shards will reveal the IDs of the $n - m$ shards that were not used. In a recovery scenario, the shards that correspond to these IDs can be removed from the set of unused shards, thereby reducing the remaining solution space. It is important to note these hints are primarily useful *after* a block has been reconstructed; less than m of n shards contain *no* information, and the hints themselves are only present in the reconstructed block.

4.3 Experimentation and Discussion

My experiments with POTSHARDS were designed to explore both the system, and the novel security model that I have developed. First, I wanted to evaluate the performance of the system in order to establish its effectiveness, and to identify any potential bottlenecks. Second, I wanted to demonstrate the ability of POTSHARDS to recover from a lost archive. Third, I wanted to demonstrate the effectiveness of approximate pointers, and understand their behavior. Finally, I wanted to explore the unique security model of POTSHARDS.

The current version of POTSHARDS consists of roughly 1,400 lines of Python version 2.5 code. For improved buffer management, versus standard Python lists, SciPy version 1.1.0 arrays were utilized extensively. Further, while most of the current version is implemented in native Python with SciPy code, an exception was the threshold secret splitting scheme. For this, I utilized an optimized C library, developed by Kevin M. Greenan, that in-

Splitting Parameters first split, second split	Ingestion (MB/s)	Extraction (MB/s)
(1, 1, null), (1, 1, null)	46.00	16.70
(1, 1, null), (3, 3, XOR)	26.18	16.26
(1, 1, null), (2, 3, Shamir)	8.09	9.25
(1, 1, null), (3, 4, Shamir)	5.05	8.05
(2, 2, XOR), (2, 3, Shamir)	4.17	6.12

Table 4.1: Ingestion and extraction performance for a variety of configurations. The splitting parameters are expressed in tuples of the form $(m, n, \texttt{algorithm}$ where the first tuple corresponds to the first split, and the second tuple to the second split. For testing, a pass-through algorithm named "null" was created which appends metadata but does no secret splitting.

cludes a $GF(2^8)$ arithmetic based implementation of Shamir's linear interpolation algorithm [67].

All of my experiments were performed on identical hardware, and were the only processes running aside from basic system processes. Each host was equipped with four dual-core AMD Opteron™ 2212 processors with 8 GB of RAM and ran Linux 2.6.18-92.el5.

During these experiments, the data transformation component utilized object sizes of 750 KB. Since POTSHARDS is designed for archival storage, block sizes are expected to be relatively large, on the order of a few hundred kilobytes to a megabyte, and possibly larger. Additionally, the default approximate pointer width, *R*, was 30. Unless otherwise noted, the first layer of secret splitting used an XOR based algorithm and produced two fragments per object, and the second layer utilized a 2 of 3 Shamir threshold scheme. The workloads consisted of randomly generated files, all larger than 1MB in size. While these files are representative of the files that a long-term archive might contain, it is important to note that POTSHARDS sees all objects as the same, regardless of the object's origin or content.

4.3.1 Read and Write Performance

My first set of experiments evaluated the ingestion and extraction performance of the POTSHARDS client. Table 4.1 presents the throughput of a single POTSHARDS client at various parameters. A workload of randomly selected academic literature totaling 25 MB was selected as it provided stable throughput numbers, and reflects the type of data likely to be encountered by an archival system.

In order to establish a performance upper-bound for the client operations, I created a pass-through algorithm that did no secret splitting but left all other client operations — such as metadata processing and index generation — intact. The results with this "null" splitter, seen in the first line of Table 4.1, show that extraction lags considerably behind ingestion. This is largely a factor of system write caching.

With an upper bound established, my goal was to measure the performance of the secret splitting operations. As Table 4.1 shows, simple XOR splitting is considerably faster than the compute intensive Shamir algorithm. For reference, in isolated tests, my optimized Shamir implementation achieved secret splitting throughput of 7.6 MB/s, and a secret combining throughput of 19.3 MB/s with a 3 of 5 split. Extraction times with *m* of *n* secret splitting algorithms are often faster than ingestion times for two reasons. First, ingestion involves the overhead of generating random data for the secret splitting algorithms. Second, secret regeneration in the extraction process begins as soon as sufficient shares have been obtained; reconstruction does not need to wait for all *n* shares.

Finally, Table 4.1 shows the client throughput with a first level XOR split and a second level Shamir split, the "default" POTSHARDS configuration. This arrangement demonstrated the slowest throughput rates, although this is to be expected for a number of reasons. First, with two levels of secret splitting, there are two levels incurring a random data generation penalty. Second, and more importantly, with an initial $(2, 2, \texttt{xor})$ split, followed by a $(2, 3, \texttt{Shamir})$ split, the second level splitter is splitting over twice as much data as the user had submitted. Further in my experiments, system throughput is measured from the user's perspective; demands inside the system are six times those seen by the client.

4.3.2 Archive Reconstruction

The archive recovery mechanisms were run on the local system using eight 1.5 GB archives. Each redundancy group in the experiment contained eight archives encoded using RAID 5. A 25 MB client workload was ingested into the system using 2 of 2 XOR splitting and 2 of 3 Shamir splitting, resulting in 150 MB of client shards, excluding the appropriate parity. After the workload was ingested, an archive was failed. I then used a static recovery manager that sent reconstruction requests to all of the available archives and waited for successful responses from a fail-over archive. Once the procedure completed, the contents of the failed

Secrets	10	20	50
(2,3)	37.08	94.28	698.89
(2,4)	68.01	202.42	1523.41
(3,4)	1872.14	10305.86	180080.86

Table 4.2: Recovery time, in seconds, for a variety of secret splitting parameters using the brute-force approach in which approximate pointers are not used.

archive and the reconstructed archive were compared. This procedure was run three times, recovering at 14.5 MB/s, with the verification proving successful on each trial. The procedure was also run with faults injected into the recovery process to ensure that the verification process was correct.

4.3.3 User Data Recovery

In the absence of approximate pointers, reconstructing data from a set of shards is a difficult combinatorics problem. Lacking any outside information, each shard must be matched with every other shard and a reconstruction attempt must be made on every chain of length *m*. Approximate pointers enable the reconstruction of user data in a reasonable time. The experiments of this section were designed to explore the difference between the various recovery heuristics, and to understand how different naming and splitting parameters affect recovery.

4.3.3.1 Recovery Heuristics

In other to establish a recovery baseline, a pure combinatorics approach of attempting reconstruction on every combination of *m* shard was attempted. This strategy, while still time consuming, takes advantage of two aids. First, the shard names provide at least a partial ordering. Second, the appended hash can confirm a successful reconstruction. As expected, the results of Table 4.2 shows that this approach does not scale beyond a handful of shards, and serves only as a baseline or last resort recovery strategy.

With a baseline established, I evaluated usefulness of approximate pointers with both the naïve and the ring heuristic described in Section 4.2.5. While user indices provide for efficient read and write performance under most access scenarios, Figure 4.10 shows that approxi-

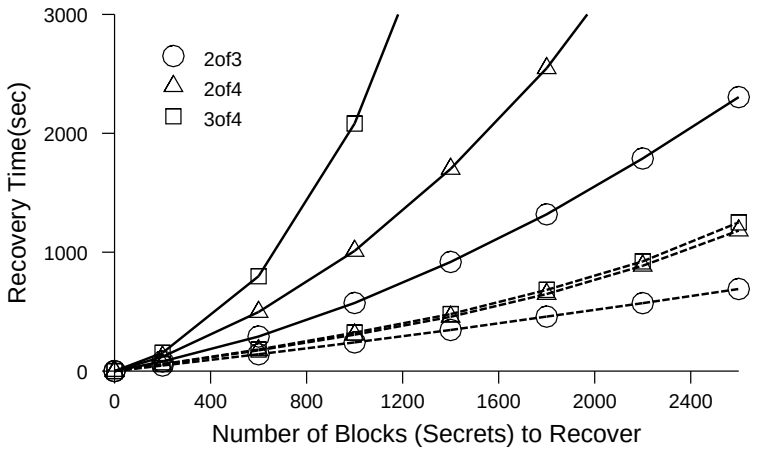
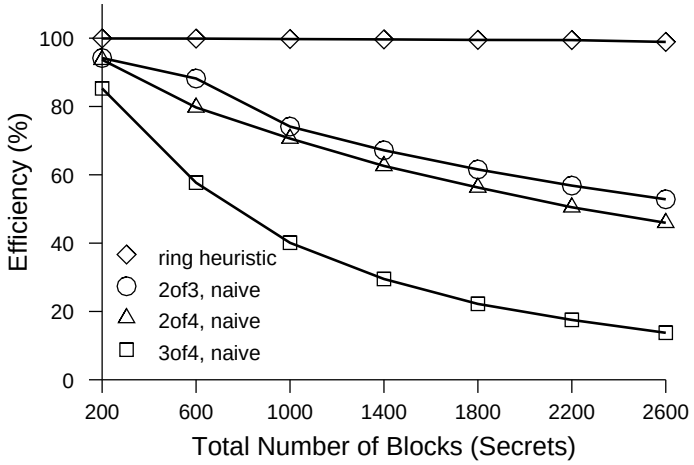


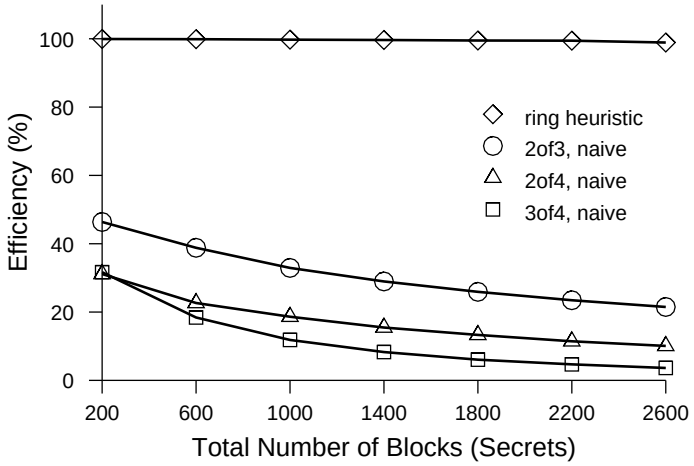
Figure 4.10: Recovery time, in seconds, for various values of *m* and *n* with both the naïve approach and the ring heuristic. Reconstruction plots that use the ring heuristic are shown using a dashed line.

mate pointers can provide adequate recovery performance when an index is unavailable. As the number of shards increases, the ring heuristic provides dramatically faster recovery times when compared to the naïve approach, and both are orders of magnitude faster than the approach that does not use approximate pointers, as shown in Table 4.2. This is quite apparent when comparing the recovery times for data resulting from a 3 of 4 split. The ring heuristic was able to recover 2,600 secrets in 1,251 seconds; in contrast, the naïve approach took 17,712 seconds. Even this, however, is an improvement compared to the brute-force approach which required over 180,000 seconds to recover just fifty secrets.

Recovery times are largely computationally limited because *m* of *n* threshold schemes often rely upon expensive operations. Thus, in addition to recovery times, I can also measure the efficiency of the strategies based on how often they select false shard tuples. By this definition, perfect efficiency would be achieved if every shard tuple selected reconstructed a valid secret. Figure 4.11(a) shows the comparison of three different secret splitting settings and their recovery efficiency. From my experiments, two things are evident. First, the ring heuristic is very efficient at selecting valid shard tuples with all three of the secret splitting settings. Second, larger values of *m* adversely affect the efficiency of the naïve approach. This is due to the fact that as *m* increases, the number of paths of length *m* increases greatly. Given a shard with an approximate pointer that points to *R* candidate shards, and a namespace density of $D = (0, 1]$,



(a) Share Tuple Selection Efficiency



(b) Shamir Call Efficiency

Figure 4.11: Efficiency of different recovery strategies as the total number of shards increases. Efficiency of shard tuple selection is the percentage of tuples selected by the recovery heuristic that reconstruct a valid secret. Efficiency of the Shamir call is the percentage of reconstruction attempts that reconstruct a valid secret. The ring heuristic was used with all three secret splitting parameter settings and each gave similar results. Thus, all of the results obtained using the ring heuristic were averaged and shown as one plot in order to improve clarity.

there are $(RD)^{m-1}$ possible paths. Thus, on average, $\frac{1}{2}(RD)^{m-1}$ paths must be tested by the naïve approach.

The difference between the ring heuristic and the naïve approach is even more pronounced when the efficiency of the secret splitting operation is measured. Figure 4.11(b) clearly illustrates two important points. First, the ring heuristic benefits from a full shard tuple and thus a total ordering over the secret shares. Therefore, each potential shard tuple selected by the ring heuristic only needs to be tested by the Shamir reconstruction operation once. Figure 4.11 shows the result; the efficiency of the ring heuristic is the same at the shard tuple selection level as it is at the secret splitting operation level. In contrast, with only m of the total n secret shares, the shard names only provide a partial ordering. Thus, a shard tuple selected by the naïve scheme must be tested by the secret splitting reconstruction operation up to $\frac{n!}{m!(n-m)!}$ times before it can be confirmed as invalid.

It might be tempting to believe the ring heuristic provides an additional layer of data secrecy because a user with only a partial set of shards is unable to utilize the ring heuristic to its full potential. However, it is important to bear in mind that once an intruder has enough shards to reconstruct data, security is only computationally bound; subsequently, it must be assumed that it is only a matter of time until data is revealed. Thus, the system's goal is to survive long enough and make attacks noticeable enough to prevent an adversary from acquiring sufficient shards to computationally recover plaintext blocks.

4.3.3.2 Population

The *population* of an approximate pointer can be described as the number of valid shards indicated by each approximate pointer and is closely tied to the width of the approximate pointer. For example, a well-formed traditional pointer would have a population of one shard per pointer and a null pointer has (rather appropriately) a population of zero shards. Further, suppose an approximate pointer p indicates a region $[p-2, p+2]$. If there are three shards in this range, $p-2, p+1, p+2$, the density of p is 0.6. Managing population is important because, if it is too high, it will be more difficult to detect intruders and will negatively affects recovery times. On the other end of the spectrum, if the number of shares per approximate pointer is too low, an unacceptable portion of the namespace is being wasted.

The density of a region, as calculated by dividing the population, P, of a region by its

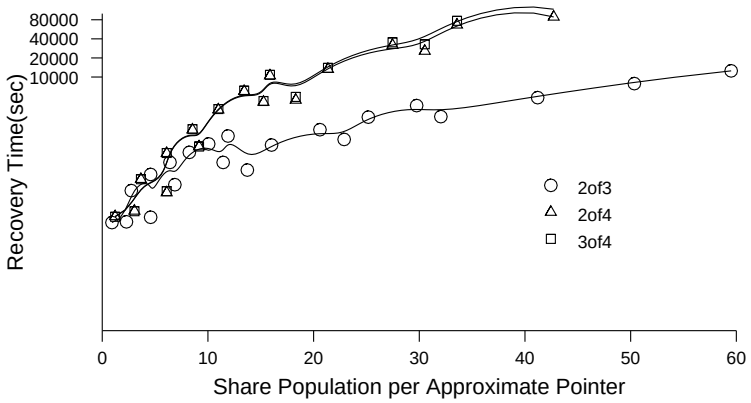


Figure 4.12: The effect of the approximate pointers' populations on the time to recover 2,600 secrets using the ring heuristic. In these tests, population, P, was modified by adjusting the size of the region, R, indicated by the approximate pointer; density was kept constant.

size, R, affects the ease with which malicious data accesses can be detected. Suppose a fictional adversary has obtained a shard and is requesting additional shards based on the approximate pointer. Assuming the attacker is restricted to making one request at a time, there are a number of possible outcomes of a shard request. First, there is a chance, approximately $1-P/R$, that the attacker will make an *invalid guess* by requesting a shard that does not exist (name assignment within a region is random, and hence the number of valid shards in a region may not be precisely P). This property is integral to the use of approximate pointers with a sparse namespace because this outcome is very noticeable by an archive, which can log the invalid access. Second, there is the chance that a malicious attacker will successfully make a *correct guess*. In this scenario, correctness is defined as successfully requesting the shard that actually belongs to the same shard tuple as their current shard. Third, there is a chance that the attacker can make a *valid guess*. If a guess is valid, then there is an actual shard at the requested address, but it does not belong to the same shard tuple as the attacker's shard. Thus, all correct guesses are valid guesses, but the reverse is not true. Both correct and valid guesses are difficult to use in detecting attackers because normal users as well as attackers make them. However, invalid guesses are much more often unique to attackers because normal users will typically know exactly which shards they need and not request nonexistent shards.

The population of an approximate pointer also has an effect on data recovery times. Even with the ring heuristic, recovering objects from shards, when faced with no other outside

information, amounts to controlling a combinatorics problem of exponential growth. This is evident in Figure 4.12 which shows the recovery time for 2,600 secrets at various population levels per pointer. Population was increased by modifying the width, R, of the approximate pointers; the shard density was constant. The tests were run utilizing the ring heuristic and, as would be expected, the tests that required cycles of length four to be tested grew at a faster rate than those that only had to test cycles of length three.

4.3.4 Security Model

4.3.4.1 Secret Splitting Parameters

The secret splitting parameters used greatly affect many aspects of the system's security including data leakage, recovery times and efficiency. The three aspects of secret splitting parameter selection include the values of m, n and the difference between the two, $n - m$.

Higher values of m provide a higher level of data protection, but can also lead to higher recovery times. As Figure 4.13 and Figure 4.14 illustrate, less data was leaked when larger values of m were used. However, there is the risk that recovery times will be higher if less than the full shard tuple can be acquired. While approximate pointers and the naïve approach are still useful, Figure 4.10 and Figure 4.11 demonstrate that higher values of m incur a penalty for recoveries with less than a full set of n shards. This scenario can, however, be mitigated in two ways. First, a hybrid solution can be utilized in which as many secrets as possible are recovered using the ring heuristic. Then, the remaining shards can be recovered using the naïve approach. Second, as Figure 4.6 illustrates, the list of shard identifiers that a fragment generates is appended to the fragment. Thus, upon successful reconstruction of a fragment from only m shards, the remaining shards can be identified and removed from the list of unused shards. This reduces the solution space for the subsequent secret recoveries. The results of my experiments suggest that larger values of m should be chosen when secrecy is a priority over potential recovery times.

In an m of n threshold scheme, the value of n directly impacts the storage overhead and in turn the namespace density. One technique for managing the namespace relies on careful name allocation. Entities that draw security directly from their position in the namespace, such as shards that rely on noticeable attacks, should be placed sparsely. In contrast, entities that do not draw their security from their position in a namespace can be densely packed. For example,

when performing a two-layer split, the identifiers for the original data and the identifiers for the results of the first split can all be drawn from a small, densely packed portion of the total namespace. The majority of the namespace, however, can be sparsely populated and devoted to shard names.

Despite their increased namespace overhead, higher values of n do provide security benefits. As Figure 4.13 shows, higher values of n can be useful for limiting the amount of information leaked, albeit mostly as a result of allowing higher values of m. To this end, the experimentation suggests that a higher value of n, along with a correspondingly higher value of m, provides the most protection against lost data.

4.3.4.2 Risk of Data Compromise

While secret splitting and approximate pointers are designed to make attempts to steal specific shards easy to detect and survive, there is also the possibility that a large scale compromise could occur. This could occur in a scenario in which shards are stored in a distributed manner across several data stores. If some of those archives are either compromised or collude to reconstruct data, there is the possibility of data being revealed.

To determine the amount of data that can be revealed from a large scale compromise, and to better understand how to limit it, I measured the data that could be regenerated from a random subset of secret shares. In my experiment, 2,600 secrets were split using Shamir's linear interpolation scheme. From the resulting set of secret shares, an increasing percentage was randomly selected and as much data as possible was recovered. The results, shown in Figure 4.13, indicate two things. First, less data is released when both m and n increase and $n - m$ is held constant. In my experiment, using a 3 of 4 split revealed the least amount of information. Second, for a fixed m, increasing n reveals an increasing amount of information. This is not unexpected as the odds of randomly selecting a secret share from a given tuple increase as the size of the tuple increases. In fact, threshold schemes are often used because of the availability that can be achieved by increasing the value of n.

One approach to minimizing data loss from large scale compromises is the two-level secret splitting technique used by POTSHARDS. To test the benefits of this strategy, I utilized an initial n of n XOR based split. Each of the resulting shares is then split using Shamir's m of n threshold scheme. The results, for two different values of n at the XOR split, are shown in

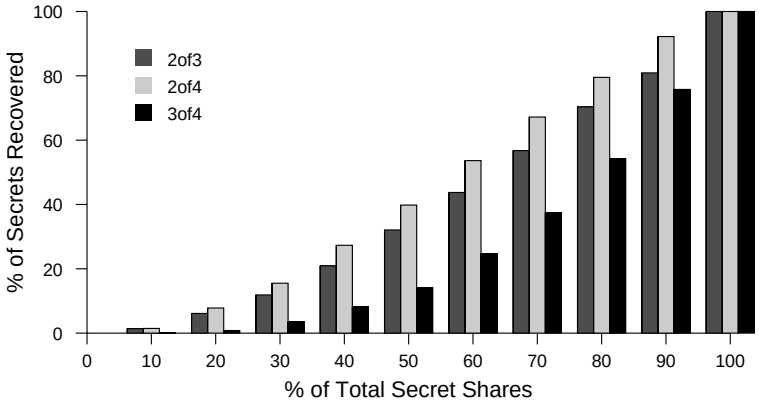
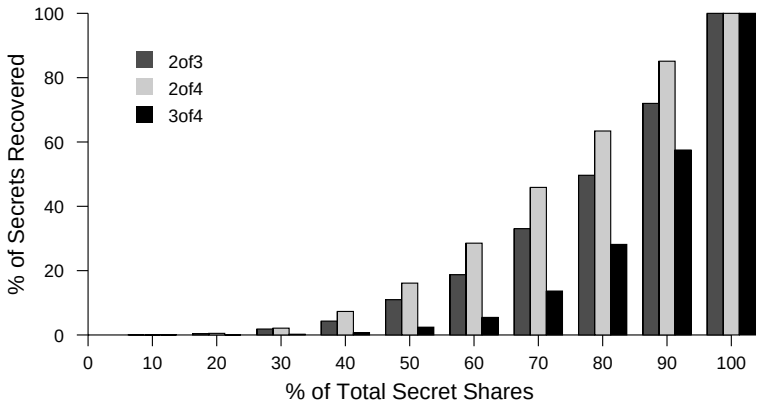


Figure 4.13: Percentage of 2,600 total blocks (secrets) that could be recovered by an adversary in a large-scale compromise. Tests were performed over randomly selected, partial sets of secret shares.

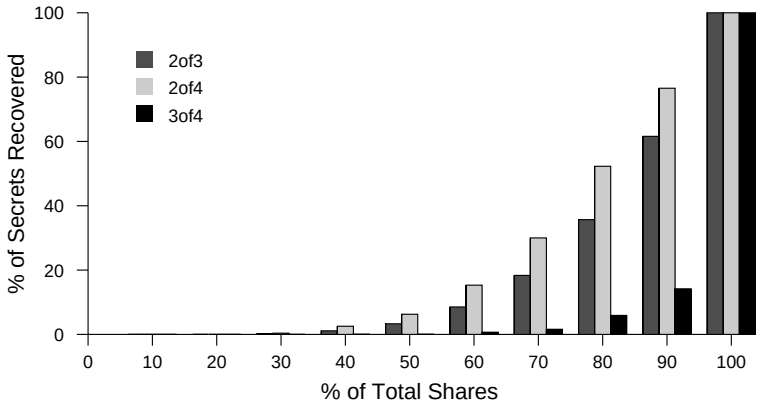
Figure 4.14 and indicate that the additional level of secret splitting is effective at lowering the amount of information released. Also, as in the single split, larger values of $n-m$ at the second layer of secret splitting still resulted in higher amounts of information loss.

Our experiments also indicate that a larger split at the first level of splitting further limited the amount of information loss. This is evident in Figure 4.14(b), which shows that revealing 20% of the total number of secret shares under a first level 3 of 3 split revealed no data, regardless of the second-level split. The same 20% compromise with a first level split of two and second level 2 of 3 or 2 of 4 split resulted in 0.42% and 0.50% of the total number of secrets being revealed, respectively. Even with 60% of the total number of secret shares and a 3 of 4 Shamir split, a first level split of three only revealed an average of 0.68% of the secrets. In contrast, with 60% of the secret shares, a first level split of two revealed 5.46% and the single layer of splitting alone revealed 24.69%. Of course, 60% of the total number of secret shares represents a very large scale compromise—over half of the shares stored for the user have been acquired; I expect that compromises are more likely to result in 10% or fewer shares being acquired, given the intrusion detection approaches made possible by sparse namespaces.

Of course, while my results do show that multiple layers of secret splitting enhance security, they do incur a storage penalty. As Figure 4.6 shows, there is already a constant amount of storage overhead in the form of hashes and identifiers. These costs are, however, dominated by the storage blowup intrinsic to secret splitting. This situation is exacerbated by multiple



(a) Top Level XOR Split of Two



(b) Top Level XOR Split of Three

Figure 4.14: Percentage of 1,300 total blocks (secrets) that could be recovered by an adversary in a large-scale compromise. Tests were performed over randomly selected, partial sets of secret shares in which secrets were guarded through two levels of secret splitting: a top level XOR split and a lower level threshold split.

levels of splitting. For example, a first level split of three along, with a second 3 of 4 split incurs a storage blowup of twelve.

A system that distributes secret shares to multiple archives can further limit data loss through careful share distribution. In my experiments, all secret shares were pooled and reconstruction was attempted on a random subset of those shares. In a storage model where shares are distributed to independent archives, a more likely scenario of large scale compromise is for an adversary to acquire all of the shares on a single archive. In this situation, rather than compromising a random subset of shares, the compromise would be a specific subset: shares that reside on the compromised archives. To this end, careful distribution of shares to archives could further limit data loss.

4.3.4.3 Chaff Shards

When a shard that does not exist is requested, either mistakenly or due to a malicious user, there are two possible responses: an error message or a chaff shard. The use of chaff [21, 140] (fake packets) has been suggested as an approach to providing data secrecy without encryption. A key difference, however, is that the "chaffing and winnowing" strategy uses chaff as its primary secrecy mechanism. In the model that I am investigating, secrecy comes primarily from secret splitting. Thus, in my model, when a request is made for a shard that does not exist, a seemingly valid chaff shard is generated and returned to the user.

The primary security advantage of chaff is that the attacker is not alerted that the request for a false shard has been detected. This is not unlike a silent alarm that alerts authorities without raising the suspicion of the intruder. Thus, the role of chaff is not to slow down recovery time. In a scenario where a malicious user has obtained sufficient shards, it is only a matter of time before the data is revealed regardless of the existence of chaff. Data secrecy, whether from encryption or secret sharing, is reducible to a computationally-bound problem once an intruder has acquired enough ciphertext. Thus, the existence of chaff shards is similar to an increased key size in that it makes the problem more difficult but it does not fundamentally change the potential for data exposure.

There are two possible strategies for dealing with a user that requests a shard multiple times in order to test its validity; if a shard is requested twice, but the returned result is different each time, it is clear to the user (or attacker) that the shard is simply chaff. First, chaff can

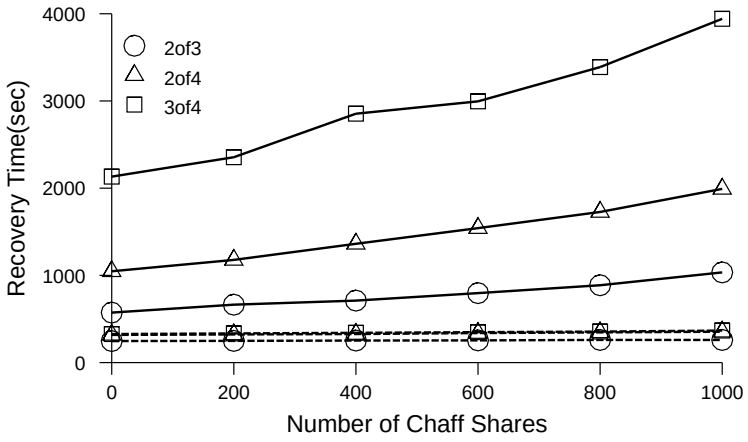


Figure 4.15: The effect of chaff on the time to recovery 1000 secrets. Recovery attempts that utilized the ring heuristic are shown using a dashed line.

be generated using a deterministic process. Alternatively, the chaff can be generated randomly and then stored. One issue with the second strategy is that a user could attempt to intentionally request false shards in order to pollute a user's set of shards. The aim of such an attack might be to use the increased recovery time as a type of denial of service attack. In fact, as Figure 4.15 demonstrates, chaff does not dramatically increase the recovery time, especially if the user is able to utilize the ring heuristic.

4.4 Publication History and Status

The publication history of POTSHARDS covers a wide gamut. Early on in the project, two papers were published that document the system's development. The earliest published work for the project appeared in 2005, at the Security in Storage Workshop (SISW) [163]. This paper represents early design ideas, and has been largely supplanted by subsequent publications. Second, in 2006, a short workshop paper was presented at the Storage Security and Survivability (StorageSS) workshop that outlined many of the security threats that POTSHARDS was designed to guard against [162].

The primary POTSHARDS paper appeared in 2007, at the USENIX Annual Technical Conference [165]. The results reported in this paper were based on a Java based implementation that suffered from poor performance, but still demonstrated the design's feasibility and merit.

In an effort to realize better performance, the entire system was rewritten using Python and C for computation intensive operations. In addition to producing better performance numbers, this system was also used to perform an in depth evaluation of the system's security. Results gathered from this implementation, along with the latest design ideas have been documented, and submitted to the ACM's Transactions on Storage (TOS) journal. Pending minor revision, it is due to be published in early 2009.

Long-term security remains a relatively new area, and the design of POTSHARDS includes a number of areas that would benefit from further study. Currently, POTSHARDS depends on strong authentication and intrusion detection to keep data safe, but it is not clear how to defend against intrusions that may occur over many years, even if such attacks are detected. One potential approach would be to refactor data so that partial progress in an intrusion can be erased by making new shards "incompatible" with old shards [196]. Unlike the failure of an encryption algorithm, which would necessitate wholesale re-encryption, refactoring for security could be done over time to limit the window over which a slow attack could succeed. Refactoring could also be applicable to secure migration of data to new storage devices.

Another area of improvement that would increase the feasibility of POTSHARDS would be a reduction in storage overhead. Some information dispersal algorithms may have lower overheads than Shamir secret splitting. Assuming that the system's information-theoretic security properties can be maintained, these algorithms may prove useful.

4.5 Conclusion

This chapter discussed POTSHARDS, a system designed to provide secure long-term archival storage to address the new challenges and new security threats posed by archives that must securely preserve data for decades or longer. The goal is to create a security model that relies not on a large key-space, but on surviving attacks and making attacks easy to detect and respond to.

Experiences with an early implementation show that users can store data at over 4 MB/s and retrieve user data over 6 MB/s. Since POTSHARDS is an archival storage system, throughput is more of a concern than latency, even these unoptimized throughputs exceed typical long-term data creation rates for most environments.

Experiments also show that the ring heuristic is effective at recovering data from

even a large set of secret shares. From an efficiency standpoint, the total ordering that the ring heuristic imposes over a potential shard tuple provides a dramatic improvement compared to the naïve approach of testing only paths of length *m*. Additionally, I demonstrate that increasing *m* and utilizing multiple levels of secret splitting can minimize the amount of data revealed in the event of a large scale data compromise. My experiments also show that chaff shares do not dramatically increase recovery times. Thus, their benefit is primarily to act as a silent alarm, which does not alert an adversary that they have been detected.

By addressing the long-term threats to archival data while providing reasonable performance, POTSHARDS provides reliable data protection specifically designed for the unique challenges of secure archival storage; the use of secret splitting, a sparse namespace and approximate pointers are well suited to the unique secrecy and recovery demands of archival data with a potentially indefinite lifetime. Storing data in POTSHARDS ensures not only that it will remain available for decades to come, but also that it will remain secure and can be recovered by authorized users even if all indexing is lost.

Chapter 5

Energy-Efficient, Archival Storage

> Yet it is in our idleness, in our dreams, that the submerged truth sometimes comes to the top
>
> Virginia Woolf

An area of scalability largely overlooked by traditional system is the ability to scale over time [18, 19]. With Pergamum, I demonstrate my thesis statement — archival storage is a first class storage category that requires solutions tailored for long-lived data — by describing a system designed specifically for efficient, long-term data storage. Unlike traditional systems, Pergamum favors evolvability and cost efficiency over absolute performance, a design choice that is valid for archival data.

Pergamum introduces several new techniques to disk-based archival storage. First, my system distributes control to the individual devices, rather than centralizing it, by including a low-power CPU and network interface on each disk; this approach reduces power consumption by eliminating the need for power-hungry servers and RAID controllers. Systems such as TickerTAIP [26] used distributed control in a RAID, but did not include reliability checking and power management. Second, Pergamum aggressively ensures data reliability using two forms of redundancy: intra-disk and inter-disk. In the former, each disk stores a small number of redundancy blocks with each set of data blocks, providing a self-sufficient way of recovering from latent sector errors [16, 17]. In the latter, Pergamum computes redundancy information across multiple disks to guard against whole disk failure. However, unlike existing RAID systems, Pergamum can stagger inter-disk activity during data recovery, minimizing peak energy

consumption during rebuilding. Third, energy-efficient decentralized integrity verification is enabled by storing data signatures for disk contents in NVRAM. Thus, using just the signatures, Pergamum tomes can verify the integrity of their local contents and, by exchanging signatures with other Pergamum tomes, verify the integrity of distributed data without incurring any spin up costs. Finally, the Pergamum architecture allows disk-based archives to look like tape: an individual Pergamum tome may be pulled out of the system and read independently; the remaining Pergamum tomes will eventually treat this event like a disk failure and rebuild the "missing" data in a new location.

The goal of Pergamum, is to realize significant cost savings by keeping the vast majority, as many as 95%, of the disks spun down while still providing reasonable performance and excellent reliability. My techniques allow Pergamum to greatly reduce energy usage, as compared to traditional hard drive based systems, making it suitable for archival storage. The use of signatures to verify data reduces the need to power disks on, as does the reduced scrubbing frequency made possible by the extra safety provided by intra-disk parity. Similarly, staggering disk rebuilds reduces peak power load, again allowing Pergamum to reduce the maximum number of disks that must be active at the same time. While I believe these techniques are best realized in a distributed system such as Pergamum—the use of many low-power CPUs is more efficient than a few high-power servers—they are also suitable for use in more conventional MAID architectures, and could be used to reduce power consumption in them as well.

5.1 System Components

The design of Pergamum was driven by a workload that exhibits read, write and delete behavior that differs from typical disk-based workloads, providing both challenges and opportunities. The workload is write-heavy, motivated by regulatory compliance and the desire to save any data that *might* be valuable at a later date. Reads, while relatively infrequent, are often part of a query or audit and thus are likely to be temporally related. Deletes are also likely to exhibit a temporal relationship as retention policies often specify a maximum data lifetime. This workload resembles traditional archival storage workloads [127, 202], adding deletion for regulatory compliance.

The Pergamum system is structured as a distributed network of independent storage appliances, as shown in Figure 5.1. Alone, each Pergamum tome acts as an intelligent storage

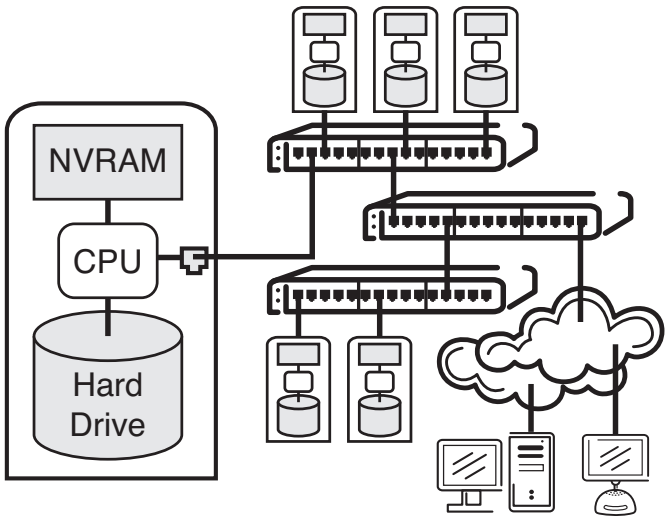


Figure 5.1: High-level system design of Pergamum. Individual Pergamum tomes, described in Section 5.1.1 are connected by a commodity network built from off-the-shelf switches.

device, utilizing block-level erasure coding to survive media faults and algebraic signatures to verify block integrity. Collectively, the storage appliances provide data reliability through distributed RAID techniques that allow the system to recover from the loss of a device, and inter-disk data integrity by efficiently exchanging hash trees of algebraic signatures. As I will show, this approach is so reliable that disk scrubbing [149] need not be done more frequently than annually. In addition, lost data can be rebuilt with lower peak energy consumption by staggering disk activity; this approach is slower, but reduces peak power consumption.

The next two sections discuss the design and implementation of Pergamum and implementation of these techniques. This section describes an individual Pergamum appliance, or *tome*, including its components, intra-appliance redundancy strategy, interconnection network, and interface. Section 5.2 then describes how multiple storage appliances work together to provide reliable, distributed, archival storage, including a description of the system's inter-appliance redundancy and consistency checking strategy.

5.1.1 Pergamum Tomes

A Pergamum tome is a storage appliance made up of four main components: a low-power processor, a commodity hard drive, non-volatile flash memory and an Ethernet controller. To protect against media errors, erasure coding techniques are used on both the hard drive and

Component	Power
SATA Hard Drive [191]	7.5 W
ARM-based board (w/ NIC) [15]	3.5 W
NVRAM	< 0.6 W

Table 5.1: Active power consumption (in watts) of the four primary components that make up a Pergamum tome.

flash memory.

Each Pergamum tome is managed by an on-board, low-power CPU; a modern ARM-based single board computer consumes 2–3 W when active (using a 400 MHz CPU) and less than 300 mW when inactive [15]. The processor handles the usual roles required of a network-attached storage device [60, 61] such as network communications, request handling, metadata management, and caching. In addition, each Pergamum tome's CPU manages consistency checking and parity operations for the local drive, responds to search requests, and initiates communications with other disks to provide inter-disk reliability. The processor can also be used to handle other operations at the device level, such as virus checking and compression.

Persistent storage is provided through the unit's SATA-class hard drive. The use of commodity hardware offers cost savings over more costly SCSI and FC drives while providing acceptable performance for archival workloads. By using both intra-disk redundancy and distributed redundancy groups, commodity SATA-class drives can provide excellent reliability for long-term archival storage [147].

While a single processor could manage multiple hard drives, Pergamum pairs each processor with a single hard drive. This is done for performance matching, power savings, and ease of maintenance. As Section 5.3 details, low-power processors are not fast enough to run even a single disk at full speed, so there is little incentive to control multiple disks with a single CPU. Power savings is another issue: a faster CPU and multi-disk controller would consume more power than multiple individual low-power CPUs (cutting processor voltage in half results in half the clock speed but one fourth the power consumption). Finally, the pairing of a CPU with a single disk and network connection makes it simpler to replace a failed Pergamum tome. If any part of the Pergamum tome fails, the entire Pergamum tome is discarded and replaced, rather than trying to diagnose which part of the Pergamum tome failed to "save" working hard

drives. The system then heals itself by rebuilding the data from the failed device elsewhere in the system. By reducing the complexity of routine maintenance, Pergamum reduces ongoing costs.

In addition to a hard drive, each Pergamum tome includes a pool of on-board NVRAM for storing metadata such as the device's index, data signatures and information about pending writes. The purpose of the NVRAM is to provide low-power, persistent storage; operations such as metadata searches and signature requests do not require the unit's drive to be spun up. While the use of flash-type NVRAM provides better persistency and energy-efficiency compared to DRAM, it does raise two issues: reliability and durability. Pergamum tome protects the flash memory from erroneous writes and media errors through the use of page-level protection and consistency checking [66], ensuring memory reliability. Flash memory is also limited in that the memory must be written in blocks, and each block may only be rewritten a finite number of times, typically 10^4–10^5 times. However, since the NVRAM primarily holds metadata such as algebraic signatures and index information, flash writes are relatively rare; flash writes coincide with disk writes. Because this typically occurs fewer than 1000 times per year, or 8000 times during the lifetime of a disk, even if the flash memory is totally overwritten each time, such activity will still be below the 10,000 write cycles that flash memory can support. Additionally, while the current implementation uses NAND flash memory, other technologies such as MRAM [173] and phase change RAM [30] could be used as they become available and price-competitive, further reducing or eliminating the rewrite issue.

Finally, each Pergamum tome includes an Ethernet controller and network port, providing a number of important advantages. First, a network connection is a standardized interface that changes very slowly—modern Ethernet-based systems can interoperate with systems that are more than fifteen years old. In contrast, tape-based systems require a unique head unit for each tape format, and each of those devices may require a different interface; supporting legacy tapes could require the preservation of lengthy hardware chains. The use of a network also eliminates the need for robotics hardware (or humans) to load and unload media; such robots might need to be modified for different generations of tape media and must be maintained. Instead, the system can use commodity network interconnects, leaving all media permanently connected and always available for messaging.

5.1.2 Interconnection Network

Since Pergamum must contain thousands of disks to contain the petabytes of data that long-term archives must hold, its network must scale to such sizes. However, throughput is not a major issue for such a network—a modern tape silo with 6,000 tapes typically has fewer than one hundred tape drives, each of which can read or write at about 50 MB/s, for an aggregate throughput of 5 GB/s. Scaling a gigabit Ethernet network to support comparable bandwidth can be done using a star-type network with commodity switches at the "leaves" of the network and, potentially, higher-performance switches in the core. For example, a system built from 48-port gigabit Ethernet switches could use two switches as hubs for 48 switches, each of which supports 46 disks, with the remaining two connections going to each of the two hubs. This approach would support over 2200 disks at minimal cost; if the central hubs each had a few 10 Gb/s uplinks, a single client could easily achieve bandwidth above 5 GB/s. This structure could then be replicated and interconnected using a more expensive 10 Gb/s switch, allowing reasonable-speed access to any one of tens of thousands of drives, with the vast majority remaining asleep to conserve power.

The interconnection network must allow any disk to connect with any network-connected client. By using a standard Ethernet-based network running IP, Pergamum ensures that *any* disk can communicate with any other disk, allowing the system to both detect newly-connected disks and allowing them to communicate with existing disks to "back up" their own data.

The approach described above is highly scalable, with minimal "startup cost" and low incremental cost for adding additional disks. Further efficiencies could be achieved by pairing the Ethernet cable with a higher-gauge wire capable of distributing the 14–18 W that a spun-up disk and processor combination requires. Alternatively, the system could use disks that can spin at variable speeds as low as 5400 RPM [191], reducing disk power requirements to 7.5 W and overall system power needs to below 11 W, sufficiently low to use standard power-over-Ethernet. Central distribution of power has several advantages, including lower hardware cost and lower cabling cost. Additionally, distributing power via Ethernet greatly simplifies maintenance—adding a new drive simply requires plugging it into an Ethernet cable. While the disks in the system will work to keep average power load below 5% utilization, a central power distribution system will allow the network switches themselves to guarantee that a particular power load will never be exceeded by restricting power distributed by the switch.

5.1.3 Pergamum Tome Interface

There are two distinct data views in Pergamum: a file-centric view and a block-centric view. Clients utilize the file-centric view, submitting requests to a Pergamum tome through traditional read and write operations. In contrast, requests from one Pergamum tome to another utilize the block-centric view of data based on redundancy group identifiers and offsets.

Clients access data on a Pergamum tome using a set of simple commands and a connection-oriented request and response protocol. Currently, clients address their commands to a specific device, although future versions of Pergamum will include a self-routing communications mechanism. Internally, files are named by a file identifier that is unique within the scope of a single Pergamum tome. The `new` command allocates an unused file identifier and maps it to a filename supplied by the user. This mapping is used by the `open` command to provide the file's unique identifier to a client. This file id, the device's `read` and `write` commands, and a byte offset are then used by the client to access their data.

Requests between Pergamum tomes primarily utilize a data view based on segment identifiers and block offsets, as opposed to files. There are four main operations that take place between Pergamum tomes. First, external parity update requests provide the Pergamum tome storing parity with the delta and metadata needed to update its external redundancy data. Second, signature requests are used to confirm data integrity. Third, token passing operations assist in determining which devices to spin up. Finally, there are commands for the deferred (*foster*) write operations discussed in Section 5.2.3.1.

Management of Pergamum tomes can be done either with a centralized "console" to which each Pergamum tome reports its status, or in a distributed fashion where individual Pergamum tomes report their health via LED. For example, each Pergamum tome could have a small green LED that is on when the appliance is working correctly, and off when it is not. An operator would then replace Pergamum tomes whose light is off; this approach is simple and requires little operator skill. Alternatively, a central console could report "Pergamum tome 53 has failed," triggering a human to replace the failed unit. The Pergamum design permits both approaches.

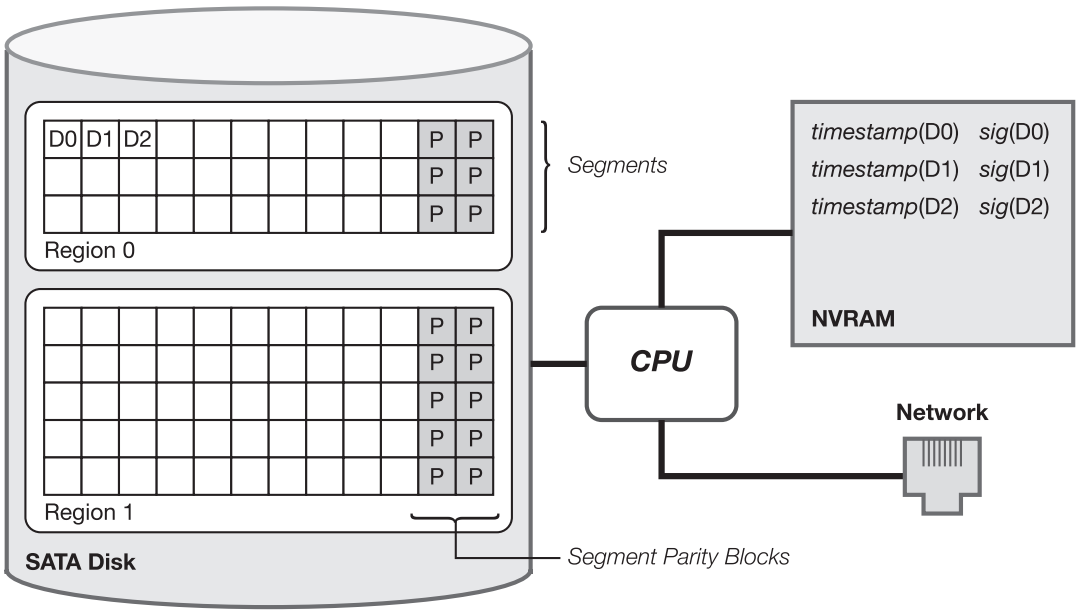


Figure 5.2: Layout of data on a single Pergamum tome. Data on the disk is divided into blocks and grouped into segments and regions. Data validity is maintained using signatures, and parity blocks are available to rebuild lost or corrupted data.

5.2 Pergamum Algorithms and Operation

A Pergamum system, deployed as described in Section 5.1 is highly decentralized, relying upon individual disks to each manage their own behavior and their own data. Each disk is responsible for ensuring the reliability of the data it stores, using both local redundancy information and storage on other nodes.

5.2.1 Intra-Disk Storage and Redundancy

The basic unit of storage in a Pergamum tome are fixed-size blocks grouped into fixed-size *segments*, as shown in Figure 5.2. Together, blocks and segments form the basic units of the system's two levels of redundancy encoding: intra-disk and inter-disk. Since the system is designed for archival storage, blocks are relatively large—128 KB–1 MB or larger—reducing the metadata overhead necessary to store and index them. This approach mirrors that of tape-based systems, which typically require data to be stored in large blocks to ensure high efficiency and reasonable performance.

The validity of individual blocks is checked using hashes; if a block's content does not match its hash, it can be identified as incorrect; this approach has been used in other file systems [94, 170]. Disks themselves maintain error-correcting codes, but such codes are insufficiently accurate for long-term archival storage because they have a silent failure rate of about 10^{-14}, a rate sufficiently high to cause data corruption in large-scale long-term storage. To avoid this problem, each disk appliance stores both a hash value and a timestamp for each block on disk. Assuming a 64-bit hash value and a 32-bit timestamp, a 1 TB disk will require 96 MB of flash memory to maintain this data for 128 KB blocks. Keeping this information in flash memory has several advantages. First, it ensures that block validity information has a different failure mode from the data itself, reducing the likelihood that both data and signature will be corrupted. More importantly, however, it allows the Pergamum tome to access the signatures and timestamps without powering on the disk, enabling Pergamum to conduct inter-disk consistency checks without powering on individual disks.

The hash values used in Pergamum are *algebraic signatures*—hash values that are highly sensitive to small changes in data, but, unlike SHA-1 and RIPEMD, are not cryptographically secure. Algebraic signatures are ideally suited to use in Pergamum because, for many redundancy codes, they exhibit the same relationships that the underlying data does. For example, for simple parity:

$$d_0 \oplus d_1 \cdots \oplus d_{n-1} = p \Longrightarrow \mathrm{sig}(d_0) \oplus \mathrm{sig}(d_1) \cdots \oplus \mathrm{sig}(d_{n-1}) = \mathrm{sig}(p)$$

While 64-bit algebraic signatures are sufficiently long to reduce the likelihood of "silent" errors to zero; they are ineffective against malicious intruders, though there are approaches to verifying erasure-coded data using signatures or fingerprints that can be used to defeat such attacks [73, 148].

As Figure 5.2 illustrates, each segment is protected by one or more parity blocks, providing two important protections to improve data survivability. First, the extra parity data provides protection against latent sector errors [16, 17]. If periodic scrubbing reveals unreadable blocks within a segment, the unreadable data can be rebuilt and written to a new block using only the parity on the local disk. Second, while simple scrubbing merely determines whether the block is readable, the use of algebraic signatures and parity blocks allows a disk to determine whether a particular block has been read back properly, catching errors that the disk drive itself cannot [73, 148] and correcting the error without the need to spin up additional disks.

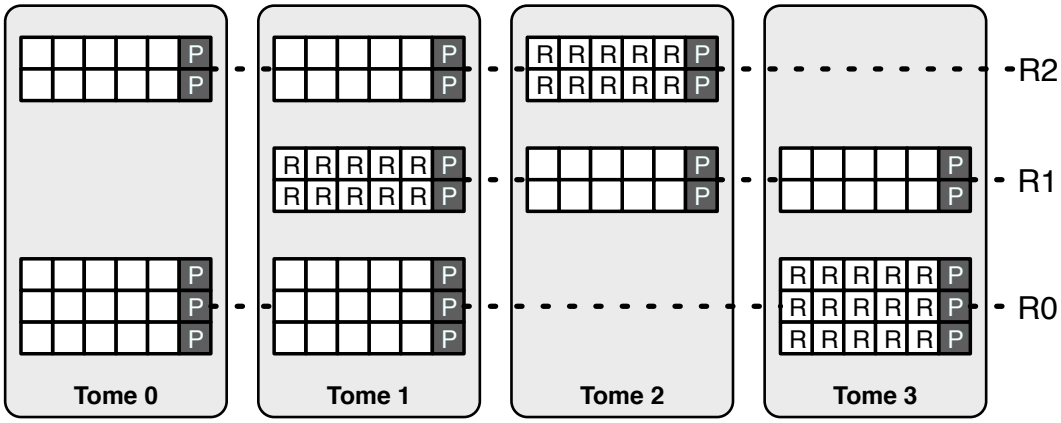


Figure 5.3: Two levels of redundancy in Pergamum. Individual segments are protected with redundant blocks on the same disk (**P**). Redundancy groups are protected by a redundancy group parity region (**R**), which contain erasure correcting codes for the other segments in the redundancy group. Note that segments used for redundancy still contain intra-disk redundant blocks to protect them from latent sector errors.

5.2.2 Inter-Disk Redundancy

While intra-disk parity guards against latent sector errors, Pergamum can survive the loss of an entire Pergamum tome through the use of inter-tome redundancy encoding. Segments on a single disk are grouped into *regions*, and a *redundancy group* is built from regions of identical sizes on multiple disks. To ensure data survival, each redundancy group also includes extra regions on additional disks that contain erasure correction information to allow data to be rebuilt if any disks fail. These *redundancy regions* are stored in the same way as data regions: they have parity blocks to guard against individual block failure and the disk appliances that host them store their algebraic signatures in NVRAM.

The naïve approach to verifying the consistency of a redundancy group would require spinning up all the disks in the group, either simultaneously or in sequence, and verifying that the data in the segments that make up the regions in the group is consistent. Pergamum dramatically reduces this overhead in two ways. First, the algebraic signatures stored in NVRAM can be exchanged between disks in a redundancy group and verified for consistency as described in Section 5.2.1. Since the signatures are retrieved from NVRAM, the disk need not be spun up during this process as long as changes to on-disk data are reflected in NVRAM. If inconsistencies are found, the timestamps may be used to decide on the appropriate fix. For example, if a

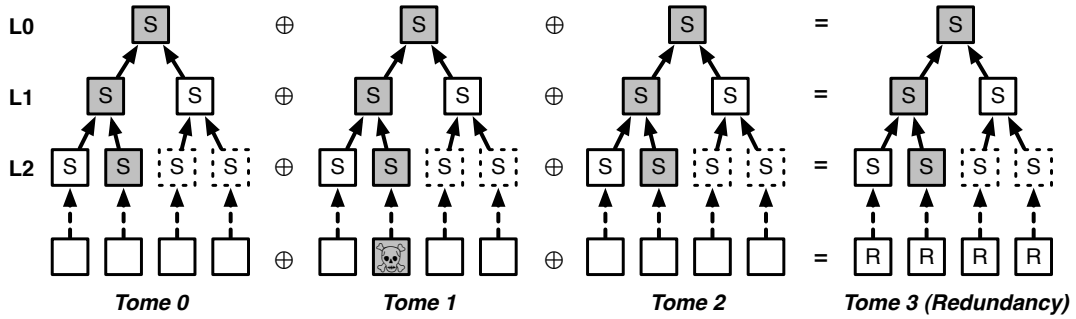


Figure 5.4: Trees of algebraic signatures. Tomes in a redundancy group exchange the roots of their trees to verify consistency; in this diagram, the signatures marked with a skull are inconsistent. The roots (L0) are exchanged; since they do not match, the nodes recurse down the tree to L1 and then L2 to find the source of the inconsistency. "Children" of consistent signatures (signatures outline with a dotted line at L2) are not fetched, saving transmission and processing time. The inconsistent block on tome 1 is found by checking the intra-segment signatures on each block; only those on tome 1 were inconsistent. Note that only tome 1's disk need be spun up to identify and correct the error if it is localized.

set of segments is inconsistent and a data segment is "newer" than the newest parity segment, the problem is likely that the write was not applied properly; depending on how writes have been applied and whether the "old" data is available, the parity may be fixed without powering up the whole set of segments.

While this approach only requires that signatures, rather than data, be transmitted, it is still very inefficient, requiring the transmission of nearly 100 MB of signatures for each disk to verify a redundancy group's consistency. To further reduce the amount of data and computation that must be done, Pergamum uses hash trees [111] built from algebraic signatures, as shown in Figure 5.4. Using signatures of blocks as d_i in Equation 5.2.1 shows that signatures of sets of signatures follow the same relationships as the underlying data; this property is maintained all the way up to the root of the tree. Thus, the signatures at the roots of each disk's hash tree for the region should yield a valid erasure code word when combined together. If they do not, some block in the redundancy group is invalid, and the disks recurse down the hash tree to find the bad block, exchanging the contents at each level to narrow the location of the "bad" block. This approach requires $O(k)$ computation and communication when the group is correct—the normal case—and $O(k \log n)$ computation and communication to find an error in a redundancy group with a total of n blocks across k disks. Since redundancy groups are not large ($k \leq 50$,

typically), high-level redundancy group verifications can be done quickly and efficiently.

5.2.3 Disk Power Management

Reducing power consumption is a key goal of Pergamum; since spinning disks are by far the largest consumer of power in a disk appliance, keeping the disk powered off ("spun down") dramatically reduces power consumption. In contrast to earlier systems that aim to keep 75% of the disks inactive [68], Pergamum tries to keep 95% or more of the disks inactive all of the time, reducing disk power consumption by a factor of five or more over existing MAID approaches. This goal is achieved with several strategies: sequentially activating disks to update redundancy information on writes, low-frequency scrubbing, and sequentially rebuilding regions on failed disks.

To guard against too many disks being spun up at once, Pergamum uses *spin-up tokens*, which are passed from one node to another to allow spin-up. If multiple nodes require a token simultaneously, the node currently holding the token (which may or may not be spun up at the time) calculates need based on factors such as a unit's oldest pending request, the types of requests it has pending, the number of pending requests and the last time the disk was spun up.

5.2.3.1 Reading and Writing Data

When a client requests a data read, the device from which data is to be read is spun up. This process takes a few seconds, after which data can be read at full speed. While a Pergamum tome is somewhat slower than a high-power network-attached disk, its performance, discussed in Section 5.3, is sufficient for archival storage retrieval. Moreover, since the data is stored on a disk rather than a tape, random access performance is significantly better than that of a tape-based system.

As with reads, archive writes require a spun-up disk. Pergamum clients choose the disks to which they write data; Pergamum does not impose a choice on users. This is done because some clients may want to group particular data on specific disks: for example, a company might choose to archive email for an individual user on one drive. On the other hand, a storage client may query Pergamum nodes to identify spun-up nodes, allowing it to select a disk that is already spun up.

Since writes require the eventual update of distributed data, they are more involved than reads. First, the target disk is spun up if it is not already active. Next, data is written to blocks on the local disk. However, existing data blocks are not overwritten in place; instead, data is written to a new data block, allowing the Pergamum tome to calculate "deltas" based on the old and new block. These deltas are then sent to the Pergamum tomes storing the redundancy regions for the old block's segment. On the local device, the segment mapping is updated to replace the old block with the new block. It is important to note, however, that the old block is retained until it has been confirmed that all external parity has been updated.

On the Pergamum tomes storing the redundancy information, the deltas arrive as a parity update request. Since the redundancy update destination knows how the erasure correcting code is calculated, it can use the delta from the data target disk to update its own redundancy information; it does not need both the old and new data block, only the delta. Because the delta may be different for different parity disks, however, the Pergamum tome that received the original write request must keep both old and new data until all of the parity segments have been updated. However, doing updates this way ensures that a write requires no more than two disks to be active at any time; while the total energy to write the data is unchanged—a write to an (m,n) redundancy group must still update $n-m+1$ disks—the peak energy is dramatically reduced from $n-m+1$ disks active to 2 disks active, resulting in an improvement for any code that can correct more than one erasure.

One problem with allowing writes directed to a specific Pergamum tome is that the disk may not be spun up when the write is issued. While the destination disk may be activated, an alternate approach is to write the data to *any* currently active disk and later copy the data to the "correct" destination [114, 115]. This approach is called *surrogate writing*, and is used in Pergamum to avoid spinning disks up too frequently. Instead, writes are directed to an already-active disk, and the Pergamum tome to which data will eventually be sent is also notified. The data can then be transferred to the correct destination lazily.

5.2.3.2 Scrubbing and Recovering Data

To ensure reliability, disks in Pergamum are occasionally scrubbed: every block on the disk is read and checked for agreement with the signature stored in NVRAM. This procedure is relatively time-consuming; even at 10 MB/s, a 1 TB disk requires more than a day to check.

However, Pergamum tome's use of on-disk redundancy to guard the data in a segment, described in Section 5.2.1, greatly reduces the danger of data loss from latent sector errors, so the system can reduce the frequency with which it performs full-disk scrubs. Instead, a Pergamum tome performs a "limited scrub" each time it is spun up, either during idle periods or immediately before the disk is spun down. This limited scrub checks a few hundred randomly-chosen locations on the disk for correctness and examines the drive's SMART status [155], ensuring that the disk is basically operating correctly. If the drive passes this check, the major concern is total drive failure, either during operation or during spin-up, as Section 5.3.2 describes.

Complete drive failures are handled by rebuilding the data on the lost drive in a new location. However, since fewer than 5% of the disks in Pergamum may be on at any given time and redundancy groups that may contain data and parity on 15–40 disks for maximal storage efficiency, it is impractical to spin up all of the disks in a redundancy group to rebuild it. Instead, Pergamum uses techniques similar to those used in writing data to recover data lost when a disk fails. The rebuilding algorithm begins by choosing a new location for the data that has been lost; this may be on an existing disk (as long as it is not already part of the redundancy region), or it may be on a newly-added disk. Pergamum then spins up the disks in the redundancy region one by one, with each disk sending its data to the node on which data is being rebuilt. The node doing the rebuilding folds the incoming data into the data already written using the redundancy algorithm; thus, it must write each location in the region m times and read it $m-1$ times (the first "read" would result in all zeros, and is skipped).

5.3 Experimental Evaluation

My experiments with the current implementation of Pergamum were designed to measure several things. First, I wanted to evaluate the system costs in order to ensure that my solution was economically feasible. Second, with the assistance of Kevin Greenan, I wanted to confirm that Pergamum can provide long-term reliability through a strategy of multiple levels of parity and consistency checking using algebraic signatures. Finally, I wanted to measure the performance of the implementation to show that Pergamum is suitable for archival workloads and to identify potential bottlenecks.

The remainder of this section proceeds as follows. First, I present an analytical evaluation of the system's cost. Then, I recap the results of Kevin Greenan's long-term reliability

simulations. Finally, I present the results of my performance tests with the current implementation of Pergamum.

5.3.1 Cost

Cost in an evolvable, long-lived system can be difficult to calculate. The advantages that a heterogeneous structure can provide by allowing a system to adapt over time, also conspire against a simple cost model of capital expenditures and operating expenditures amortized over a set accounting period. As devices are constantly arriving and leaving the systems — due to scale out, failures and evolution — all costs are on-going costs. Thus, the ideal way to measure cost would capture the utility the system provides (capacity), a unit of time, and a value (dollars).

In the absences of such detailed cost models, the traditional approach uses a straightforward strategy to calculating system cost by identifying static costs (capital expenditures), and operational costs. The first figure describes the cost to acquire the system, and the second figure quantifies the cost to run the system. Examining both costs together is important because low static costs can be overshadowed by the total cost of operating and maintaining a system over its lifetime.

I do not consider personnel costs in any of the systems I describe; I assume that all of the systems are sufficiently well automated that human maintenance costs are relatively low. However, this assumption is somewhat optimistic, especially for large tape-based systems that use complex hardware that may require repair. In contrast, Pergamum is built from simple, disposable components—a failed Pergamum tome or network switch may simply be thrown out rather than repaired, reducing the time and personnel effort required to maintain the system.

Static costs reflect the expenses associated with acquiring an archival storage solution, and can be calculated by totaling a number of individual costs. One is the system expense, which totals the base hardware and software costs of a storage system with a given capacity for storage media. This cost is paid at least once per storage system, regardless of how much storage is actually required. Media cost, in dollars per terabyte, is a second expense. Large archival storage systems may require several “base” systems; for example, an archival system that uses tape silos and robots might require one silo per 6,000 tape cartridges, even if the silo will not be filled initially.

Operational costs reflect those costs incurred by day to day operation of an archival

System	Media	Static cost	Oper. cost	Redundancy
Sun StorageTek SL8500	T10000 tape	$4,250	$60	None
EMC Centera	SATA HD	$6,600	$1,800	parity
PARAID	SCSI HD	$37,800	$1,200	RAID
Copan Revolution	SATA HD	$19,000	$250	RAID-5
RAIL	UDO2	$57,000	$225	RAID-4 (5+1)
Pergamum	SATA HD	$4,700	$50	2-level

Table 5.2: Comparison of system and operational costs for 10 PB of storage. All costs are in thousands of dollars and reflect common configurations. Operational costs were calculated assuming energy costs of $0.20/kWh (including cooling costs).

storage system. This cost can be measured using a dollars per operational period figure, normalized to the amount of storage being managed. Some of the primary contributors to a system's total operational expenses include power, cooling, floor space and management. Many of these are incurred by not only the storage system itself, but by infrastructure such as network and monitoring devices. As described above, I omit management cost, both because I assume it will be similar for different storage technologies, and because it is extremely difficult to quantify. I also omit the cost of floor space since it is highly variable depending on the location of the data center. However, an important, but often omitted, aspect of operational costs includes the expenses related to reliability: expected replacement costs for failed media and the operational cost associated with parity operations. This cost, along with power and cooling, forms the basis of the comparison of operational costs.

The static and operational costs must include the cost for any redundant hardware or storage. However, since existing solutions vary in their reliability, even within a particular technology, I have not attempted to quantify the interplay between capacity and reliability. Instead, I assume that a system that requires mirroring simply costs twice as much to purchase and run per byte as a non-redundant system. In this respect, Pergamum is very low cost: the storage overhead for a system with segments using 62 data and 2 parity blocks and redundancy groups with 13 data disks and 3 parity disks is $\frac{64}{62} \times \frac{16}{13} - 1 = 0.27$ times usable data capacity. In such a system, 1 TB of raw storage can hold 787 GB of user data.

All of these factors—static cost, operational cost, and redundancy overhead—are summarized in Table 5.2. Static costs are approximations based on publicly available hard-

ware prices. For operational costs, I have used a constant rate of $0.20/kWh for electricity to cover both the direct cost of power and the cost of cooling. Table 5.2 shows the costs for a 10 PB archive for each technology, including sufficient base systems to reach this capacity. While the costs reflected in the table are approximate, they are useful for comparative purposes. Also, I note that some systems have ranges for redundancy overhead because they can be configured in several ways to ensure sufficient reliability; I chose the least expensive reliability option for each technology. For example, the EMC Centera [69] can be used with mirroring; doing so might increase reliability, but will certainly increase total cost.

The results summarized in Table 5.2 illustrate a number of cost-related archival storage issues. First, as shown by PARAID, even energy-efficient, non-archival systems are too expensive for archival scenarios. Second, media with low storage densities can become expensive very quickly because they require a large amount of hardware to manage the high numbers of media. For example, RAIL uses UDO2 optical media that only offers 60 GB per disk and thus the system requires numerous cabinets and drives to handle the volume of media. Using off-the-shelf dual-layer DVDs, with capacity under 10 GB per disk, would reduce the media cost, but would increase the hardware cost by a factor of six because of the added media; such an approach would require 100 DVDs per terabyte, making the cost prohibitive. Third, the Copan and Centera demonstrate two different strategies for cost effective storage: lower initial costs versus lower runtime costs. Finally, it is clear that Pergamum is competitive in cost to Sun's StorageTek SL8500 system while providing functionality, such as inter-archive redundancy, that tape-based systems are unable to provide.

An understanding of the costs associated with reliability is important because it assists in matching the data to be protected with an economically efficient reliability strategy. Unfortunately, because it is largely dependent on the data itself, the economic impact of lost data is difficult to calculate. Moreover, many of the costs resulting from data loss are, at best, difficult to quantify. For example, the cost to replace data can vary from zero (don't replace it) to nearly priceless (how much is bank account data worth?). Another factor, opportunity costs, expresses the cost of lost time; every hour spent dealing with data loss is an hour that is not spent doing something else. In a professional setting, data loss may also involve mandatory disclosures that could introduce costs associated with bad publicity and fines. While I do not quantify these costs, I note that long-term archive reliability is a serious issue [19].

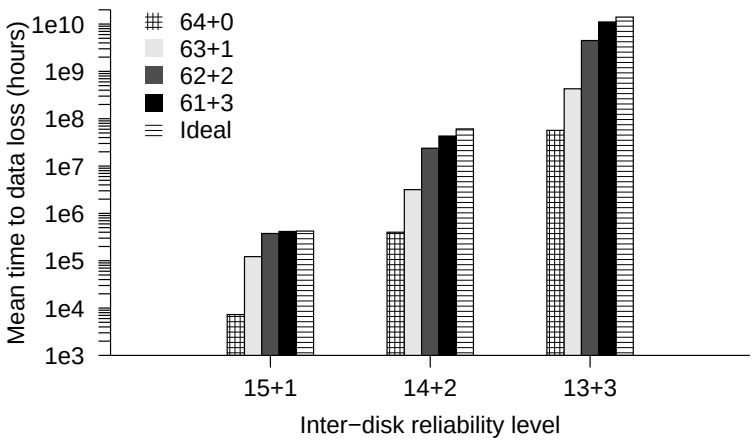


Figure 5.5: Mean time to data loss in hours for a single 16 disk group. 61+3 intra-disk parity is nearly equivalent to the "ideal" system, in which latent sectors never occur. Note that MTTDL of 10^{10} hours for 16 disks corresponds to a 1000 year MTTDL for a 10 PB Pergamum system.

5.3.2 Reliability

As in all storage, reliability is an important part of archival storage. For long-term storage, legacy systems, evolving software and migrating knowledge workers are only some of the factors that make replacing lost data difficult. In its current design, Pergamum provides two levels of reliability: intra-device and inter-device. Of course, there are many tradeoffs that influence the reliability of an archival storage system. Factors such as stripe size, both on an individual disk and between disks, disk failure rate, disk rebuild time and the expected rate of latent sector errors must be considered when building a long-term archival system.

A full exploration of the factors affecting the reliability of archival storage is beyond the scope of this work. However, the current and ongoing research of Kevin M. Greenan, shows that Pergamum is capable of providing a high degree of reliability. Summarized here (and shown in Figure 5.5), his simulation and modeling show that a configuration of 3 inter-disk parity segments per 16 disk reliability group and 3 intra-disk parity blocks per 64 block segment results in an MTTDL of approximately 10^{10} hours.

5.3.3 Performance

The current Pergamum prototype system consists of approximately 1,400 lines of Python 2.5 code, with an additional 300 lines of C code that were used to implement performance-sensitive operations such as data encoding and low-level disk operations. The implementation includes the core system functionality, including internal redundancy, external redundancy, and a client interface that allows for basic I/O interactions. In its current state however, the implementation relies upon statically assigned redundancy groups and it does not include scrubbing or consistency checking.

For testing, all systems were located on the same gigabit Ethernet switch with little outside contention for computing or network resources. Communication between the Pergamum tome and the client used standard TCP/IP sockets in Python. For maximum compatibility, I utilized an MTU size of 1500 B.

Each Pergamum tome was equipped with an ARM 9 CPU running at 400 MHz, 128 MB of DDR2 SDRAM and Linux version 2.6.12.6. The client was equipped with an Intel Core Duo processor running at 2 GHz, 2 GB of DDR2 SDRAM and OS X version 10.4.10. The primary storage on each Pergamum tome was provided by a 7200 RPM SATA drive formatted with XFS. For read and write performance experiments, I utilized block sizes of 1 MB and 64 blocks per segment. Persistent metadata storage utilized a 1 GB USB flash drive and Berkeley DB version 4.4. The workload consisted of randomly generated files, all several megabytes in size.

5.3.3.1 Read and Write Throughput

My first experiment with the Pergamum implementation was an evaluation of the device's raw data transfer performance. As Table 5.3 shows, the maximum throughput of a single TCP/IP stream to a Pergamum tome is 20 MB/s at the device. Further tests showed that, the device could copy data from a network buffer to an on-disk file at about 10 MB/s. Together, these values serve as an upper limit for the write performance that could be expected from a single client connection over TCP/IP.

Write throughput using the Pergamum software layer was tested at varying levels of write safety. The first write test was conducted with no internal or external parity updates. As shown in Table 5.3, writes without data protection ran at 4.74 MB/s. While no redundancy

Test	Client	Server
Raw Data Transfer	20.02	20.96
Raw Data Write	9.33	9.98
Unsafe Pergamum Write	4.74	4.74
XOR Parity Pergamum Write	4.72	3.25
Reed Solomon Pergamum Write	4.25	1.67
Fully Protected Pergamum Write	3.66	0.75
Pergamum Read	5.77	5.78

Table 5.3: Read and write performance for a single Pergamum tome to client connection. XOR parity writes used 63 data blocks to one parity block segments. Reed Solomon writes used 62 data blocks to two parity block segments. Fully protected writes utilize two level of Reed Solomon encoding and the server throughput reflects time to fully encode and commit internal and external parity updates.

encoding was performed in the unsafe write, the system did incur the overhead of updating segment metadata and dividing the incoming data into fixed-size blocks.

Testing with internal parity updates enabled was performed using both simple XOR-based parity and more advanced Reed Solomon encoding. In these tests, the client-side and server-side throughput differ, as Pergamum utilizes parity logging during writes. Thus, while the client views throughput as the time taken to simply ingest the data, the Pergamum tome's throughput includes the time to ingest the data and update the redundancy information. The first test utilized simple XOR-based parity in a 63+1 (63 data blocks and 1 parity block) configuration. This arrangement achieved a client-side write throughput of 4.72 MB/s and a Pergamum tome-side throughput of 3.25 MB/s. As Table 5.3 shows, using Reed Solomon in a 62+2 configuration results in similar client side throughput, 4.25 MB/s. However, the extra processing and parity block updates results in a server throughput of 1.67 MB/s.

The final write test, fully protected Pergamum tome writes, utilizes both inter- and intra-disk redundancy. Internal parity utilized Reed Solomon encoding in a 62+2 configuration. External redundancy utilized Reed Solomon with 3 data regions to 2 parity regions. In this configuration, client throughput is reduced to 3.66 MB/s as the CPU is taxed with both internal and external parity calculations. This is evident in the server throughput, which is reduced to

0.75 MB/s. However, this does reflect the time required to update both internal and external parity and thus reflects the rate at which a single Pergamum tome can protect data with full internal and external parity.

Profile data obtained from the test runs indicates the system is CPU-bound. The performance penalty for the Pergamum tome writes appears to be based largely on two factors. First, as shown in the difference between a raw write and an unsafe Pergamum tome write in Table 5.3, Python's buffer management imposes a performance penalty, an issue that could be remedied with an optimized, native implementation. Second, as seen in the difference between the XOR Pergamum tome write and the Reed Solomon write, data encoding imposes a significant penalty for lower power processors. This is further evident by the results of my read throughput tests. Since a read operation to the Pergamum tome involves less buffer management and parity operations, throughput is correspondingly faster. I was able to achieve sustained read rates of 5.78 MB/s.

While the performance numbers in Table 5.3 would be inadequate for most high-performance workloads, even the current, prototype implementation of Pergamum is capable of supporting archival workloads. For example, 1000 Pergamum tomes and a spin-up rate of only 5% can provide a system-level ingestion throughput in excess of 175 MB/s, ingesting a terabyte in 90 minutes and fully protecting it in 8 hours. At this rate such an archive built from 1 TB disks could be filled in a year.

5.3.3.2 Data Encoding

One of the primary functions of each Pergamum tome's processor is data encoding for redundancy and signature generation. Thus, I wanted to confirm that the low-power CPUs used by Pergamum to save energy are actually capable of meeting the encoding demands of archival workloads.

In my first data encoding test, I measured the throughput of the XOR operation by updating parity for 50 MB of data. I was able to achieve an average encoding rate of 20.79 MB/s on the tome's CPU. For reference, a desktop class processor using the same library was able to encode data at 201.41 MB/s. However, this performance increase comes at the cost of power consumption; the Intel Core Duo processor consumes 31 W compared to the tome's ARM-based processor which consumes roughly 2.5 W for the entire board.

Encode Operations	ARM9	Core Duo
XOR parity	20.02	201.41
Reed Solomon; 5 data, 2 parity	3.13	33.68
Data signature (64-bit)	57.44	533.33

Table 5.4: Throughput, in MB/sec, to encode 50 MB of data using the Pergamum tome's 400 MHz ARM9 board drawing 2-3 W and a desktop class 2 GHz Intel Core Duo drawing 31 W.

A similar result was achieved when updating parity for 50 MB of data protected by a 5+2 Reed Solomon configuration. As Table 5.4 summarizes, the processor on the Pergamum tome was able to encode the new parity blocks at a rate of 3.13 MB/s. For reference, the desktop processor could encode at average rate of 33.68 MB/s. Again, I notice an order of magnitude throughput increase at the cost of over an order of magnitude power consumption increase.

My final encoding experiment involved the generation of data signatures. The current implementation of Pergamum generates data signatures using $GF(2^{32})$ arithmetic in an optimized C-based library. Generating 64 bit signatures over 32 bit symbols, I achieved an average signature generation throughput of 57.44 MB/s. For reference, the same library on the desktop-class client achieved a rate of 533.33 MB/s.

My results indicate that the low-power processor on the Pergamum tome is capable of encoding data at a rate comparable to its power consumption. Additionally, I believe that it is capable of adequately encoding data for an archival system's write-once, read maybe usage model. While the current performance numbers are reasonable, my experience in designing and implementing the Pergamum prototype has shown that low-power processors greatly benefit from carefully optimized code. The early implementations provided more than adequate performance on a desktop class computer but were somewhat slow on the Pergamum tome's low-power CPU.

5.4 Publication History and Status

The publication history of Pergamum is relatively brief. Unlike POTSHARDS, which had a number of preliminary publications, the first literature on Pergamum was presented at the 2008 USENIX File and Storage Technologies (FAST) conference[168]. This was accompanied

by a solicited article in the USENIX magazine, *login*, covering the same material [167].

While the published literature on Pergamum demonstrates some of the features needed in an archival storage system, work remains to turn it into a fully effective, evolving, long-term storage system. In addition to the engineering tasks associated with optimizing the Pergamum implementation for low-power CPUs, there are a number of important research areas to examine.

The optimality of the choice of one CPU and network connection per disk is also an open question; the choice is based on both quantitative and qualitative factors, but other arrangements are certainly possible. Additionally, it has always been assumed that client machines would include modern desktop level CPUs that could be leveraged for pre-processing. Similarly, determining the best network to use to connect thousands of (mostly idle) devices is an interesting problem to consider.

Large scale, correlated failure will be inevitable with a system that numbers in the hundreds of thousands of nodes. This is largely due to the fact that interconnect failures make up a sizeable fraction of storage system unavailability [80]. Many of these, such as a failed switch causing network segmentation, are benign in the sense that data may still be safe, it is simply unreachable. However, the system's reaction in such a scenario could inadvertently cause more harm than good; the system may try and immediately rebuild all data that it could not contact. Thus, the system must be able to contend with large scale failure, especially if it results in network segmentation.

While the current design includes two levels of redundancy, intra-device and across devices, a geographic level of encoding could be very beneficial. A cross-site redundancy level could utilize distributed RAID techniques across geographically diverse installations in order to protect data from natural disasters or other "act of god" failures.

While mentioned briefly, full evaluation of an evolvable storage model requires new cost models that encompass more than simply capital and operational expenditures. Evolvable systems represent a shift away from the traditional, monolithic model of storage systems, and their costs are poorly represented by such a simple cost model. For true cradle to grave costs, the cost model should paint a complete picture of lifespans of the storage appliances in a distributed architecture. The result would be a cost model that captures the cost to produce, purchase, administer, operate, decommission, and dispose of the device.

5.5 Conclusion

This chapter discussed Pergamum, a system designed to provide reliable, cost-effective archival storage using low-power, network-attached disk appliances. Experience with an early prototype showed that Pergamum's performance is acceptable for archival storage: the use of many low-power CPUs instead of a few server-class CPUs results in disks that can transfer data at 3–5 MB/s, with faster performance possible through the use of optimized code.

More importantly, at 2–3 W/TB, Pergamum is far cheaper and more reliable than existing MAID systems, though the techniques described here may be applied to more conventional MAID designs as well. Moreover, a Pergamum system is comparable in cost and energy consumption to a large-scale tape archive, while providing much higher reliability, faster random access performance and better manageability. The combination of low power usage, low hardware cost, very high reliability, simpler management, and excellent long-term upgradeability make Pergamum a strong choice for storage in long-term data archives.

Chapter 6

Management in Evolvable Archival Systems

> Evolution is not a force but a process; not a cause but a law.
>
> John Morley

As archival storage is still a relatively unexplored area, it stands to reason there is still considerable research to be done. This chapter presents my current ideas in the area of archival storage management. It demonstrates my thesis statements — archival storage is a first class storage category that requires solutions tailored for long-lived data — by identifying the management needs specific to archival storage. In a young area of research, it is perhaps just as important to demonstrate the direction that an area of exploration is headed, as it is to document what it has already achieved.

Much of archival storage management is directed at the goal of increasing efficiency, in order to lower costs. Central to this tenet, management approaches designed for long-lived data must take into account opportunity costs in an effort to maximize efficiency, Thus hardware must be managed through its entire lifespan; a long-term management approach must facilitate nodes joining the system, manage placement of data and redundancy information, handle node failure, and gracefully phase out nodes as they age.

To this end, I have started work on Logan, a management layer that runs atop a distributed network of independent storage appliances [168]. In order to avoid any single point of failure, while each node is capable of assuming a number of administrative roles, none are required to be specialized for that role. Each device is identified by a globally unique id, and maintains a list of named attributes that describe it. Using this id and attribute name, Logan

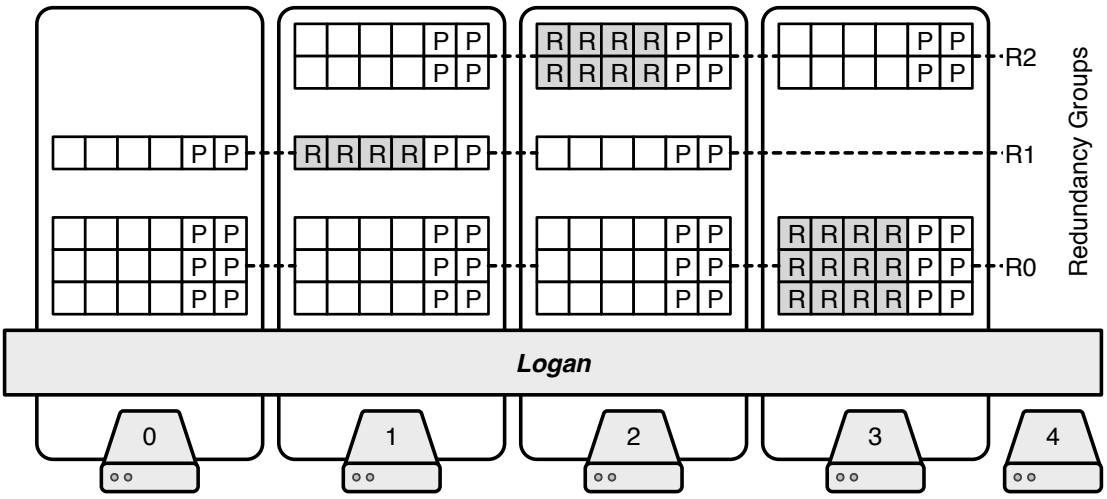


Figure 6.1: Overview of Logan running a distributed network of Pergamum tome devices [168]. One device (4) is pending inclusion in the group, while the others (0-3) are contributing to redundancy groups. Data blocks (white) are protected with internal parity (P) and external parity (R).

can query and update the node's attributes using a simple put and get interface. By updating device attributes, Logan can capture usage effects such as the accelerated wear caused by drive spin-ups, or the effects of batch correlated failures.

This decentralized, federated architecture offers a number of advantages for long-term, archival data. First, the software component of each device can act as an abstraction layer to the underlying media, enabling a heterogenous mix of technologies. Second, using multiple, low powered processors yields energy savings versus a few high powered processors (cutting processor voltage in half results in half the clock speed but one fourth the power consumption). Third, an inexpensive node can be treated as an indivisible field replaceable unit; if any part of the node fails, the entire node is discarded and replaced. This reduces the management overheard associated with locating and replacing individual components.

While each device is independent and actively ensures the longevity of its own data, nodes can cooperate in *redundancy groups* to provide system wide data reliability. Data in each device is divided into fixed sized *blocks*, and blocks are grouped into fixed sized *segments*. Distributed RAID techniques are used over groups of segments to survive the loss of a device [160]. Since heterogeneity is inevitable in an evolvable system, device capacity will vary between appliances. Thus, unlike a simple RAID system where all drives are the same size, a device can

contribute segments to more than one redundancy group in order to utilize all of its local storage capacity. Reliability can be further improved with the addition of intra-block reliability for recovering from media faults [16, 17, 43, 168].

As the global knowledge required to manage hundreds of thousands of nodes as one group would overwhelm any single data aggregation point, Logan divides devices into *management groups*. These management groups are tasked with a number of administrative duties. The first of these, *scale out*, deals with expanding the capacity of the system. It involves the creation of redundancy groups, and the assignment of segments to those groups. The second area, *recovery*, determines where data will be recovered to when a node is lost. The final area, *maintenance*, monitors the health of the system and actively identifies nodes that are ready to be decommissioned.

6.1 Design Details

The following subsection details the current design of Logan. First, I detail the hierarchical structure of the logical network. This includes a discussion of leadership election in a hierarchical network. Second, I describe how Logan manages devices from their initial installation, through their operating life.

6.1.1 Logical Structure

While modern network infrastructures allow for fully connected communication, Logan arranges the system into a logical hypercube of management groups. This topology offers a number of benefits. First, it offers efficient communications routing between management groups[177]. As Figure 6.2 shows, messages are routed in $O(\lg n)$ time by moving one address bit closer to its destination with each hop. In Logan, message routing occurs in a hierarchical fashion, with messages first routed between groups, and finally routed within the destination group. Second, hypercubes are well suited to the manner in which management groups grow and split; the exponential fan-out of hypercubes allows the system to grow to a large number of management groups, while limiting the number of edges that any single group must maintain with its neighbors.

While the inter-management group structure is that of a hypercube, the nodes within

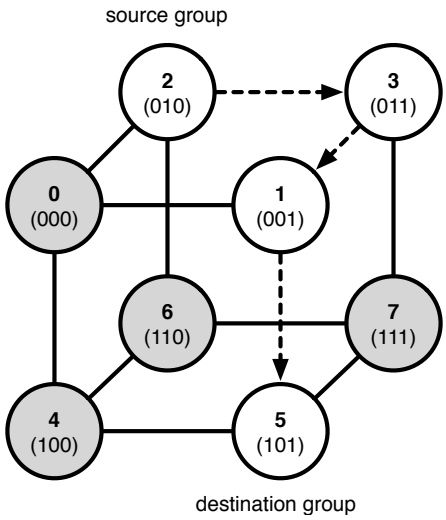


Figure 6.2: Eight management groups arranged in a hypercube of dimension 3. Nodes involved in routing from group 2 to 5 shown in white. Routing is done in $O(\lg n)$ time, since each hop brings the message one bit closer to its destination.

the management group are arranged in a logical tree. Within this tree, nodes assume one of three group roles: leader, subordinate or member. As Figure 6.3 illustrates, at the root of the tree is a *leader* that facilities operations that require a central coordinator. One degree away from the leader are one or more *subordinate* nodes. The leader and subordinates are fully connected; a bidirectional, logical edge connects the leader to each subordinate, and every pair of subordinates. Note, however, that not all nodes one degree away from the leader are necessarily subordinates. Finally, below the leader and subordinates are the remaining group members.

The intra-group structure is designed to limit the amount of logical network restructuring that occurs between elections. In the normal case, when the leader's term expires, the new leader is chosen from the subordinates, leaving the majority of the spanning tree intact, and causing little disturbance to the central clique of the leader and its subordinates. Further, as the subordinate nodes monitor the health of both each other and the leader, in the event of a leader failure, a new leader is chosen from the subordinates. In both cases, a relatively consistent group structure simplifies the management of inter-group connections.

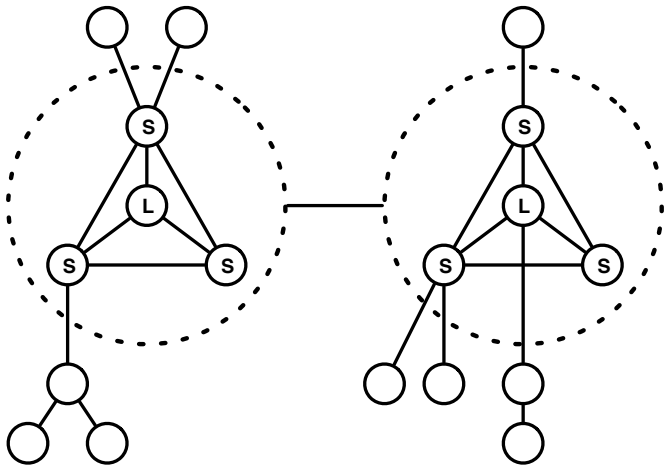


Figure 6.3: Logical structure of two groups. In each group, a single *leader* (L) forms the root of the logical tree. One degree from the leader, multiple *subordinates* (S) form a leadership clique with the leader. These are joined with the leadership cliques of other groups in the overall hierarchy. The remaining *members* of the group are arranged in acyclic, spanning trees.

6.1.1.1 System Growth

The system begins with a single management group. Nodes entering the system are added to this group until it reaches a predetermined saturation point, at which point it splits into two groups with, on average, half the membership each. Membership and splitting is based on the LH* family of distributed data structures [96–100]. Similar to a traditional hash table, LH* maps keys to buckets. However, unlike traditional hash tables which rely on a static number of buckets, LH* starts with a single bucket. As the bucket fills to capacity, it splits into two buckets. The algorithm gracefully expands from a single bucket, to an effectively unlimited number of buckets.

LH* offers a number of advantages. First, LH* does not require globally consistent data in order to function properly. This property is especially important because, in a large scale system, tight consistency expectations are unrealistic. Second, LH* is self-correcting. In the event that a client maps a key to a bucket using outdated parameters, the selected bucket will route the message to the correct bucket, as well as update the client with current parameters. Third, LH* is a light-weight protocol that does not involve computationally expensive operations, but allows the system to gracefully scale from a small system of just a single management group to a large system with thousands. Fourth, since group membership is calculated instead

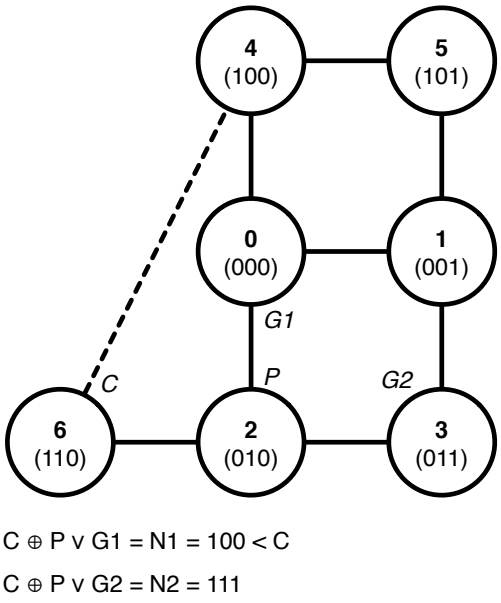


Figure 6.4: When parent P (node 2), produces a child C (node 6), it provides the child with a list of its grandparents $G1$ and $G2$ (nodes 0 and 3). The child then calculates which, if any of its bitwise neighbors it needs an introduction to from its grandparent.

of statically assigned, a node can be located from its name alone.

LH* utilizes two variables, n and i, to coordinate all of operations. For routing, n and i are parameters to the hashing function. Together with the key to map, they dictate which bucket the key maps to in the algorithm's current state. Second, n acts a token that facilitates distributed splitting; when bucket n splits, it passes the token to bucket $n+1 \bmod 2^i$. By limiting bucket splitting to the current token holder, management is simplified.

When a group splits, it must establish connections with its bitwise neighbors, N, in order to preserve the logical hypercube. As Figure 6.4 illustrates, one such connection is from the child, C, to its parent, P. This connection is easy to make. Additionally, it must establish a connection with each existing group whose name is exactly one bit different than its own name. To perform this, the parent supplies the child with a list of its grandparents, G. For each grandparent, the child calculates $C \oplus P \vee G = N$. If $N < C$, then C asks G to make an introduction. If $N \geq C$, then that bitwise neighbor has not yet been formed and no further action is required.

6.1.1.2 Management Group Leadership

The group's current leader forms the root of the logical tree and serves three primary purposes. First, as the leader is known to each member node in the group, and the leaders of adjacent groups know each other, the leader is responsible for routing messages between groups. Second, the leader is responsible for collecting information from member nodes, and using those collected statistics to determine group optimizations [102, 125, 199, 200]. Third, the leader manages redundancy group tasks such as failure recovery, maintenance, and integration of new devices.

Directly beneath the leader in the logical tree is a group of subordinate nodes. The data from the leader is replicated to the subordinates, which use the data for three primary tasks. First, in order to reduce the load the current leader, subordinates are also able to route messages between management groups. Second, as they monitor the status of the leader, they can hold an election amongst the subordinates in the event that the leader has failed. Third, when the leader's term is over, an election amongst the subordinates is held to determine the new leader.

Under most operating conditions, elections are restricted to nodes within the subordinate list. This limited election occurs in scenarios such as the end of a leader's term, or the failure of the current leader. This is done for a number of reasons. First, the subordinates already hold a replica of the leader's information. Second, by choosing a leader from the subordinate list, each election results in less overall change to the logical tree, simplifying inter-group connections. Third, as member nodes already form a spanning tree rooted at one of the central nodes, a limited election is much more efficient than a full election; after a small election restricted to the subordinates, the results can be communicated across the existing spanning trees.

As Figure 6.5 illustrates, in a limited election, three primary changes occur. First, a subordinate node, *i*, is promoted to the role of leader. Second, one of the member nodes rooted at *i* is promoted to the subordinate position. Third, the former leader becomes a member node rooted at one of the current subordinates. Once the changes have agreed upon [88], the results can be communicated to the member nodes.

In a limited election occurring at the end of a leader's term, the current leader can consult the data it has collected about its current subordinates in order decide how best to restructure the leadership clique. For example, in order to balance the spanning tree, when the

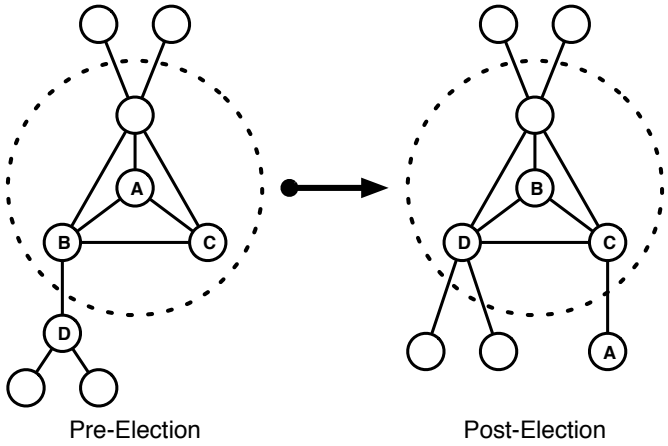


Figure 6.5: A limited election where the original leader, *A*, cedes control to a former subordinate, *B*. A former member node, *D*, is chosen by *B* to become a new subordinate, and *A* joins the tree rooted at *C* to help balance the number of nodes below each subordinate.

outgoing leader chooses a subordinate to become its parent, it can choose a node with a low number of children. For example, in Figure 6.5, node *B* was selected to become the new leader, but the outgoing leader choose to become a member rooted under node *C*.

While elections are normally restricted to the nodes within the current subordinate group, a number of scenarios may require an election over all of a group's member nodes. For example, an existing management group may split [98], and the existing leader and all of its subordinates may wind up in the same group. Other possible scenarios include bootstrapping a new network of multiple nodes, or a network segmentation that includes no leader or subordinate.

Full leadership election is a three phase process based on the construction of a spanning tree covering the nodes of the group [54, 58]. The first phase is the *election* itself, and involves a candidate node announcing their intention to become leader. The second phase is the *acknowledgement*, and it involves messages traveling back up the tree to new leader. The third and final phase is *announcement*, in which the new leader chooses its subordinate nodes, and a message with the new leader's identify travels down the newly constructed spanning tree.

Phase 1: Election Phase When a node, *L*, begins an election, it declares its intention to become the new leader by creating an election message identified by the id of the *L*. The node

L sends this message to its neighbors, and then waits for an acknowledgement from each of the nodes it has sent the message.

When a node receive an election message from node i, it takes one of two actions. First, if it recognizes that it has already received a message for this election based on the id of L, it immediately sends an acknowledgement to i. If the node has not yet participated in this election, it sets i as its parent in the spanning tree, and forwards the election message to each of its neighbors. The node then waits for an acknowledgement from node it has sent the election message.

Phase 2: Acknowledgment The acknowledgment message has two important pieces of information. First, it includes the id of L so that the acknowledgement can be associated with the correct election. Second, it includes the id of the sender's parent in the spanning tree. If the receiver is the parent indicated in the message, it adds the sending node to its list of children nodes, and removes the sender from the list of nodes from which it is awaiting an acknowledgment. If the receiver is not the parent, then it simply removes the sender from the acknowledgment waiting list. When the receiver has heard from all the nodes it has sent an election message, it then sends an acknowledgment message to its parent.

Optionally, the acknowledgement phase can also be used to collect data about the network. In this strategy, an acknowledgment message to the node's parent includes a list aggregating information for the current node, and that current node's children. In order to reduce the amount of information on the wire, acknowledgement message to non-parent nodes should not include the list of aggregated data.

Phase 3: Announcement In the final phase of the election algorithm, an announcement message travels down the newly formed spanning tree. It is in this phase that the new leader, L, names it subordinates, $S_0, \ldots, S_n$. To each of the subordinates, S_i, the announcement includes two components: the identity of the leader L, and the full list of subordinates. The first portion is used to confirm the identity of the group's leader. The second portion is used to ensure that the subordinates and leader form a clique. With this logical structure in place, the leader can proceed to replicate pertinent group data to its subordinates, and a leadership announcement can be sent along the newly constructed spanning tree.

6.1.2 Device Management

Since a system designed for long-lived data is expected to live longer than any given node, Logan enables evolution by proactively guiding each node through its entire lifespan. Figure 6.6 illustrates each device's lifecycle, starting with installation, and ending with the eventual decommissioning and removal of the device.

When a node enters first enters the system, it places itself in the NEW state, obtains an IP address and announces its presence to the system. This announcement can be done through a combination of broadcast communication, and directed communication to a distinguished node identified by an automatic configuration service such as DHCP. After it has made its announcement, as Figure 6.6 illustrates, the node enters the LONE state, and waits for a response.

In most scenarios, a node in the LONE state receives a response from an existing node, and places itself in the FLOATING state, signifying that it has made contact with system, but has yet to be integrated into its management group. The response to the new node's announcement supplies the information needed to calculate its place in the hierarchy of groups [98]. After that node has determined which group it belongs to, it asks for further assistance in routing an introduction to that group's leader. The response from the group leader will inform the node whether it is a subordinate node, or simply a member. If no response to its introduction arrives, the new node may have entered a network in the midst of a segmentation, in which case it begins a leadership election.

In a brand new system, a node in the LONE state may not receive a response because it is the first node. In this scenario, it will wait in the LONE state until is receives an announcement from the next node to join the system. In this scenario, the node initializes itself to the default values for a new installation [98], makes itself a member of the newly created management group, responds to the new node with an acknowledgement, and begins a leadership election.

Once a node has joined a management group, it is not immediately integrated into redundancy groups, but rather is placed into the PENDING state. This design provides a number of benefits. First, this allows the node to undergo a self-check and burn-in period in order to reduce the impact of infant mortality and batch correlated failures. Second, when it is time to expand the available storage in the system, Logan is able to make smarter management decisions by utilizing the devices in the PENDING pool, as compared to an approach that immediately integrates every node as soon as it arrives.

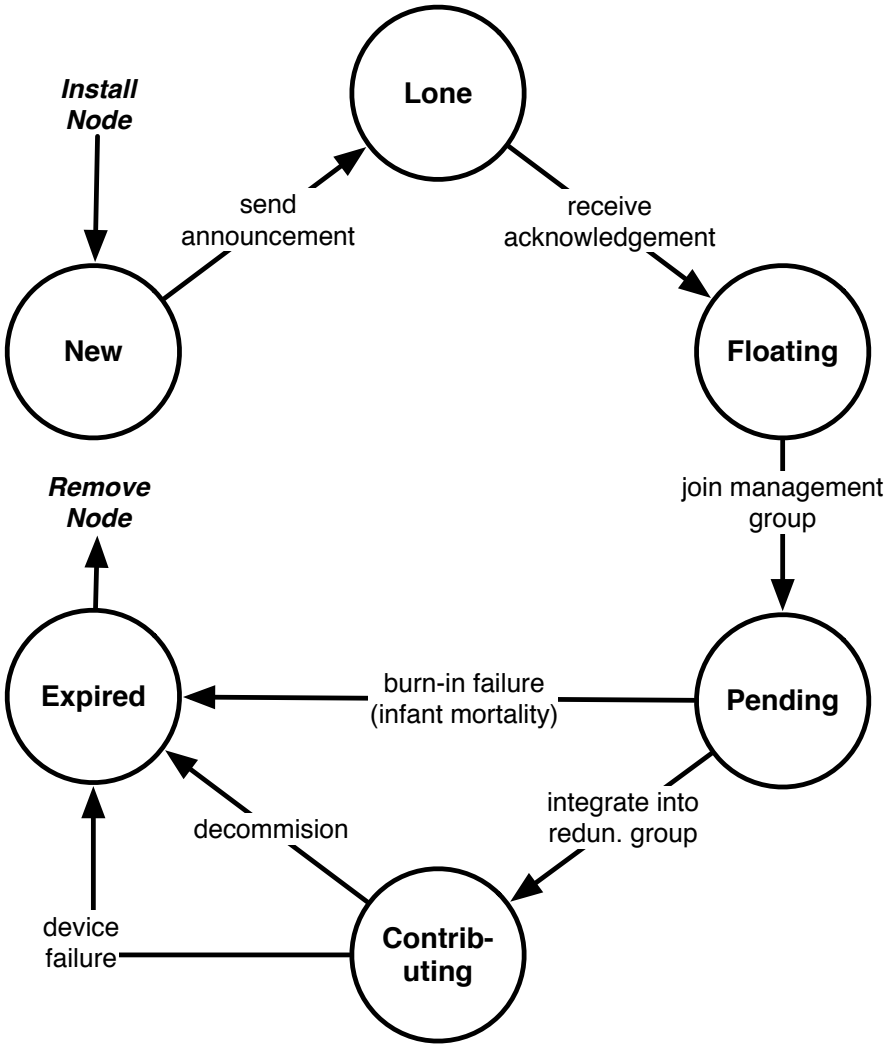


Figure 6.6: Nodes in Logan are managed through their entire lifespan, from their installation to their eventual decommissioning. Under most conditions, nodes will spend the majority of the life in the CONTRIBUTING state. In this state, the node has been integrated into one or more redundancy groups, and is actively providing resources to the system.

Once a device has entered its PENDING state, the management group leader can query the device, and update the statistics is keeps about the group's devices. This information is used during the administrative functions to identify expensive devices, in terms of utility versus resources consumed, without requiring administrator input. For example, this approach can identify a group's most power-hungry device. Further, this approach can determine how power-hungry that device is in comparison to the average of the group's devices.

In order to make good management decisions, I am exploring the use of heuristic algorithms such as simulated annealing [84]. These algorithms attempt to solve an optimization problem by utilizing heuristics to repeatedly perform minor modifications to a partial solution. To this end, these algorithms utilize three main components. First, the solution space, X is the space of all possible solutions from which the answer will be drawn. Second, the neighbor function, N, heuristically chooses a new solution that is "close" to the current solution in the solution space. Finally, an objective function, P, measures the "goodness" of a solution, and is the value that the heuristic algorithm attempts to minimize or maximize.

6.1.2.1 Scale Out

Logan monitors the system, and performs a scale out operation when it detects that available free space in a management group has dropped below a predetermined low water mark. When this occurs, the management group leader uses the information it has collected about the nodes in its group, and the existing redundancy group to decide how best to increase the amount of available storage.

For scale-out operations, each management group maintains a list of its redundancy groups and the devices assigned to those groups. This list is consulted and updated based on two redundancy group operations. First, Logan can form a new redundancy group. Second, Logan can expand an existing redundancy group. The latter strategy is possible because redundancy groups have a population range. Logan does not always fully populate new redundancy groups. Rather, it creates partially populated groups that still meet the system's reliability criteria, thus allowing the system to expand capacity, even when there are insufficient devices to create an entirely new redundancy group. For example, the system might require parity groups to be of the form $n+3$ disks, where $6 \leq n \leq 13$. This would mean that a redundancy group would have a minimum of 9 disks and a maximum of 16 disks, and be able to grow from 9 to 16 gradually

over time if needed.

At the device level, each management group maintains a list of its devices and their unassigned, or free, segments. From this pool, Logan can assign device segments to redundancy groups from two primary sources. First, Logan can utilize previously unassigned segments from a device in the CONTRIBUTING state. Second, it can utilize segments from a PENDING device. Naturally, this would cause the device to transition to the CONTRIBUTING state.

6.1.2.2 Recovery

As with any storage system, and especially a long-term archival system, failure is inevitable. Additionally, since the system must be cost efficient, it is not enough to simply recover data to the first available free space. To address this problem, Logan uses similar heuristic search techniques to determine where data should be recovered to in the event of a device failure.

For algorithms such as simulated annealing, an instance of the recovery problem solution space is a mapping of segments to redundancy groups. At each iteration of the algorithm, some subset of free segments are mapped to the segments of the failed device. The primary constraint to enforce during recovery is that each member of a redundancy group is a different device.

6.1.2.3 Maintenance

The goal of maintenance is to determine if there is a management group configuration that can offer better service for the same or lower resource consumption. For example, the amount of power required per active hard drive spindle decreases much slower than the capacity of hard drives is growing. Thus, there is an opportunity cost issue with keeping a hard drive based device in a system indefinitely. The challenge is to identify when the migration and disposal costs warrant the replacement of older devices in the pursuit of a net efficiency increase.

As in previous management group operations, the state of the system consists of a mapping of device free segments to redundancy groups. However, in the case of maintenance, the redundancy group list consists of all the existing redundancy groups. At each iteration, devices that are likely to be decommissioned based on their expected lifetime or high energy costs per segment are randomly swapped with available segments. For this operation, a valid solution enforces two constraints. First, that a device can only be decommissioned if all of its

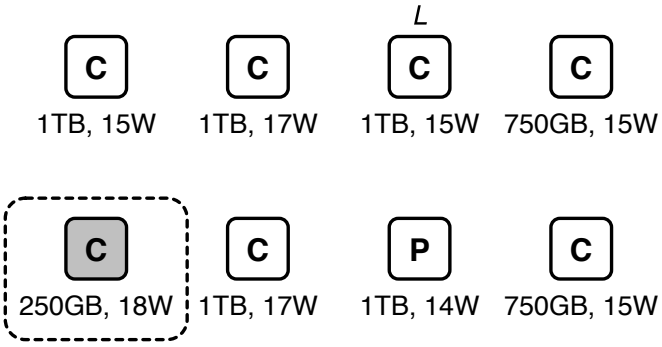


Figure 6.7: During maintenance, the current leader (L) can use the statistics it has gathered to see that one node (grey), is using a lot of power relative to the storage it offers. It can be replaced with the pending node, resulting in more available storage space, and less power consumption.

committed segments have a replacement, and that those replacements conform to the standard redundancy group constraints. Second, the total free space available after a node has been decommissioned and its segments replaced, is at least as much free space as before the device has been decommissioned.

Unlike recovery and scale-out, which are performed as soon as the heuristic completes, maintenance chores can be handled opportunistically. A device that has been identified for decommissioning can wait until a scrubbing event or recovery event occurs in order to defray the power costs associated with a wholesale migration of a nodes complete contents. An important factor that enables this opportunistic approach is that the optimizations that maintenance seeks to achieve are not critical to data safety. When the unit being decommissioned activates, it can check to see if the units slated to takes its place in redundancy groups are still available. If they are not, the decommissioning can be cancelled, or new replacements can be chosen.

6.2 Publication History and Status

Logan is, admittedly, in its formative stages, but preliminary work has been published at the 2008 Petascale Data Storage Workshop (PDSW) [164]. The literature presented indicated the direction that the management layer is taking. As the project is still relatively young, there are a number of areas to validate and explore.

Current effort on Logan is focused on refining the skeleton, described in the previous

section. Much of this work is directed towards exploring the use of heuristic algorithms in making sound management decisions. Additionally, the behavior of the system is being explored to help determine the correct size of management groups; too large and the group leaders are overwhelmed, too small and the resulting splitting results in unnecessary management overhead. Finally, I am examining the boot-strapping problem; while the system is designed to scale up to hundreds of thousands of nodes, it must inevitably start with one.

Further along in the research plans, the best way to deal with large-scale disasters and network partitions will be explored. In a long-term storage system, these sorts of events are inevitable, and must be survived gracefully, and with a minimum of needless energy expenditures. Many such events, such as a failed switch causing a network partition, are benign in the sense that data may still be safe, it is simply unreachable. However, the system's reaction in such a scenario could inadvertently cause more harm than good; the system may try and immediately rebuild all data that it could not contact.

As previously discussed, large archival systems are well suited to recovery procedures that allow the response to be scaled to the size of the problem. Currently, Pergamum utilizes a two level scheme of intra-device and inter-device reliability. A third level, across geographically diverse sites, would be useful in order to protect data from natural disasters or other "act of god" failures.

The dependency list of a given device describes the nodes that contribute to the reliability of a given node's data. Put another way, if a device fails, all of the devices in the failed device's adjacency list will need to contribute data during the recovery process. Thus, the size of the dependency list could have considerable impact on data reliability, and during recovery, energy consumption. A large redundancy group allows greater parallelization during recovery, and implies greater diversity in the redundancy group's devices. In contrast a smaller adjacency list requires less devices to spin up during recovery. Considering these and other potential tradeoffs, an understanding of how adjacency affects reliability and power consumption could allow us to tailor optimization methods to their ideal size.

Another intersection of reliability and power can be seen in a failed devices recovery schedule. That is, the amount and ordering of parallelization that occurs during rebuild. With a fuller understanding of power use during rebuild Logan could determine not only the placement of recovered data, but also the order that recovery should proceed. This area is complicated by

the affect of very transient system states. For example, device population changes much slower than the list of currently spun up devices.

Currently, the election algorithm relies on each node acting correctly and altruistically. In a real system, it may be useful for the algorithm to be resistant to malicious collusion. Taken a step further, the election algorithm could be extended to be Byzantine fault tolerant in the face of nodes that behave incorrectly [27].

Another deficiency of this election strategy is rooted in the mostly random nature of the current algorithm; the node that begins the election is essentially nominating itself. Further, the subordinates are automatically assigned; the subordinate nodes themselves have no control over whether they are selected or not. In an effort to better accommodate a gradually evolving system, the algorithm could be extended to take node characteristics into account for both nominations, and subordinate selection [178]. Ideally, the system would choose the most capable and healthiest nodes to burden with extra responsibility.

6.3 Conclusion

This chapter presented my current work in the area of archival management. While long-lived systems are well served by a distributed architecture, such a design introduces the management challenges of heterogeneity in an evolving and aging system. Further, as part of a comprehensive cost strategy, such a system should continuously seek ways to maximize the utility it offers for the resources it is consuming. This requires a system that can manage a node through its entire lifespan; unlike a traditional system, archival management must be able to integrate new devices with minimal administrator input, and device removal cannot hinge on failure or wholesale system removal.

Chapter 7

Conclusion

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> Winston Churchill

My thesis demonstrated measurable progress in the area of long-term archival research, and indicates where the current research is headed. Specifically, my work focused on the security, cost-efficiency and management of evolvable archival storage. Of course, as the area is still rather young, there is considerable work still to accomplish. To that end, this chapter proceeds as follows. First, I present some new directions for future archival storage research (specific work is discussed in the relevant chapter). Finally, I recap the contributions made by the three systems I developed: POTSHARDS for long-term security and recoverability, Pergamum for cost-efficiency and reliability, and Logan for management and evolvability.

7.1 Future Work

While my research has been concerned primarily with preserving the bits that make up files; understanding the bits is an orthogonal problem that must also be solved. Others have begun to address this problem [63], but maintaining the semantic meanings of bits over decades-long periods may prove to be an even more difficult problem than securely maintaining the bits themselves.

While POTSHARDS presents several approaches to long-term data secrecy and recoverability, there are many more security properties that rely on strategies ill-suited for long-

term assurance. While some work has been done on providing improving integrity and accountability in archival systems, long-term security is still a relatively young area [176].

Quite often, laboratory-based projects in storage research are testbeds for exploring a very focused problem. For example, POTSHARDS was developed to explore data secrecy and recoverability in long-term scenarios. Of course, customers tend to want multiple features, and many common storage techniques are incompatible when combined in an ad hoc manner. To that end, exploring the interactions between mechanisms could be a fruitful area of research. For example, while some progress has been made in exploring deduplication and security [161], there is considerable work to be done. Similarly, as with security mechanisms, deduplication may be at loggerheads with a number of energy conservation techniques; if data chunks are spread across multiple, idle disks, a data read may involve spinning up a number of devices.

One of the most valuable resources for long-term storage would be experience with a large, long-term storage system in a real-world archiving scenario. This would provide two important bodies of research information. First, filesystem traces would provide useful workload specifics that could help guide low-level design details. Second, at a higher level, user and administrator input could help validate the assumptions made about how archival storage is used; disruptive technology is often utilized in scenarios distinct from its intended application. In this respect, archival storage still feels like a young topic. In contrast to other areas of storage research, long-term storage is still largely guided by conjecture and assumptions.

7.2 Conclusion

Businesses and consumers are becoming increasingly conscious of the value of archival data. In the business arena, data preservation is often mandated by law [2, 3], and data mining has proven to be a boon in shaping business strategy. For individuals, a shift has occurred in how cultural histories are recorded. The artifacts of our personal narratives – photos, videos, correspondences, legal and medical records – are all being created and stored as digital information. Unfortunately, traditional storage systems are not designed to meet the needs of long-term, archival data [18, 19].

I have shown in my thesis that archival storage is a first-class storage category that requires solutions specifically tailored for data with an indefinite lifespan. POTSHARDS demonstrated that many common assumptions, such as the effectiveness of cryptography, are invalid

in long-term scenarios. Pergamum demonstrated that considerable energy efficiency can be achieved by exploiting the different access patterns of data, and metadata, while still providing very high levels of reliability. Finally, Logan established the need for aggressive system maximization, and the need for administration that can automatically manage a device through its entire lifespan; the great paradox with archival storage lies in the inverse relation between the value of archival data and need to aggressively pursue cost efficiency.

In developing POTSHARDS, I made several key contributions to secure long-term data archival. First, the use of multiple layers of secret splitting, approximate pointers, and archives located in independent authorization domains to ensure secrecy, shifts security of long-lived data away from a reliance on encryption. The combination of secret splitting and approximate pointers forces an attacker to steal an exponential number of shares in order to reconstitute a single fragment of user data; because he does not know which particular shares are needed, he must obtain *all* of the possibly-required shares. Second, I demonstrated that a user's data can be rebuilt in a relatively short time from the stored shares *only* if sufficiently many pieces can be acquired. Even a sizable (but incomplete) fraction of the stored pieces from a subset of the archives will not leak information, ensuring that data stored in POTSHARDS will remain secret. Third, with approximate pointers and a sparse namespace, intrusion detection is made easier by dramatically increasing the amount of information that an attacker would have to steal, and requiring a relatively unusual access pattern to mount the attack. Fourth, long-term data integrity is ensured through the use of RAID algorithms across multiple archives, allowing POTSHARDS to utilize heterogeneous storage systems with the ability to recover from failed or defunct archives and a facility to migrate data to newer storage devices.

The novel architecture of Pergamum featured several advancements and demonstrated the feasibility of a distributed design consisting of low-power, intelligent storage appliances. The two-level reliability model of Pergamum allows the response to the scaled to the size of the problem: intra-disk redundancy allows an individual device to automatically rebuild data in the event of small-scale data corruption, while inter-disk redundancy provides protection from the loss of an entire device. Fixed costs are kept low through the use of a standardized network interface, and commodity hardware such as SATA drives; since each Pergamum tome is essentially "disposable", a system operator can simply throw away faulty nodes. Operational costs are controlled by utilizing ultra-low-power CPUs, power-managed disks and new techniques

such as local NVRAM for caching metadata and redundancy information to avoid disk spin-ups, intra-disk redundancy, staggered data rebuilding, and hash trees of algebraic signatures for distributed consistency checking.

Logan, while still in a relatively formative stage, lays the groundwork for a management layer that runs atop, a distributed network of energy-efficient, intelligent storage appliances [168]. Nodes are arranged in redundancy groups, which allows data to be recovered from a lost node. To manage redundancy groups, and to facilitate system-wide communication, Logan arranges devices into management groups. Further, Logan collects information about the nodes in each management group and uses this data to make intelligent management decisions. Logan helps control archival storage costs by automating a number of common administrative tasks, and opportunistically decommissioning old hardware.

Bibliography

[1] Historical notes about the cost of hard drive storage space. `http://www.littletechshoppe.com/ns1625/winchest.html`, January 2008.

[2] 104th Congress, United States of America. Public law 104-191: Health Information Portability and Accountability act (HIPPA), October 1996.

[3] 107th Congress, United States of America. Public law 107-204: Sarbanes-oxley act of 2002, February 2002.

[4] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 2005)*, pages 59–72, San Francisco, CA, USA, December 2005.

[5] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Early experiences on the journey towards self-* storage. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, September 2006.

[6] Ittai Abraham and Danny Dolev. Asynchronous resource discovery. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (PODC 2003)*, pages 143–150, Boston, MA, USA, July 2003.

[7] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA, December 2002.

[8] Sandip Agarwala, Arnab Paul, Umakishore Ramachandran, and Karsten Schwan. e-safe: An extensible, secure and fault tolerant storage system. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 257–268, Boston, MA, USA, July 2007. IEEE Computer Society.

[9] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Anderson, Mike Burrows, Timothy Mann, and Chandramohan A. Thekkath. Block-level security for network-attached disks. In *Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 159–174, San Francisco, CA, USA, April 2003.

[10] David G. Anderson, Jason Franklin, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. Technical Report CMU-PDL-08-108, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, USA, May 2008.

[11] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the FAST 2002 Conference on File and Storage Technology (FAST 2002)*, pages 175–188, Monterey, CA, USA, January 2002.

[12] Ross Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communication Security*, pages 215–227, Fairfax, VA, USA, November 1993.

[13] Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *Proceedings of the International Workshop on Information Hiding (IWIH 1998)*, pages 73–82, Portland, OR, USA, April 1998.

[14] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked*

Systems Design & Implementation (NSDI 2005), pages 129–142, Boston, MA, USA, May 2005.

[15] Arcom, Inc. `http://www.arcom.com/`, August 2007.

[16] Lakshmi N. Bairavasundaram, Garth Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, pages 223–238, San Jose, CA, USA, February 2008.

[17] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2007)*, pages 289–300, San Diego, CA, USA, June 2007. ACM Press.

[18] Mary Baker, Kimberly Keeton, and Sean Martin. Why traditional storage systems don't help us save stuff forever. In *Proceedings of the First IEEE Workshop on Hot Topics in System Dependability (HOTDEP 2005)*, Yokohama, Japan, June 2005.

[19] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopolous, Petros Maniatis, TJ Guili, and Prashanth Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of Eurosys 2006*, pages 221–234, Leuven, Belgium, April 2006.

[20] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 197–210, San Francisco, CA, USA, March 2004.

[21] Mihir Bellare and Alexandra Boldyreva. The security of chaffing and winnowing. In *Advances in Cryptology - ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 2000. Proceedings*, volume 1976/2000 of *Lecure Notes in Computer Science*, pages 517–530. Springer Berlin, 2000.

[22] Howard Besser. Digital longevity. *Handbook for Digital Projects: A Management Tool for Preservation and Access*, pages 155–166, 2000.

[23] William J. Bolosky and Scott Corbin. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium (WinsSys 2000)*, Seattle, WA, USA, August 2000.

[24] Stewart Brand. *The Clock of the Long Now*. Basic Books, New York, NY, USA, 1999.

[25] Anne Broache. Hard drive vanishes from va facility. *CNET News.com*, February 2007.

[26] Pei Cao, Swee Boon Lin, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems (TOCS)*, 12(3):236–269, August 1994.

[27] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault tolerant system. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI 2004)*, San Diego, CA, USA, October 2004.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium of Operating Systems Design and Implementation (OSDI 2006)*, pages 205–218, Seattle, WA, USA, November 2006.

[29] Fay Chang, Minwen Ji, Shun-Tak Leung, John MacCormick, Sharon Perl, and Li Zhang. Myriad: Cost-effective disaster tolerance. In *Proceedings of the FAST 2002 Conference on File and Storage Technology (FAST 2002)*, Monterey, CA, USA, January 2002. USENIX Association.

[30] Y. C. Chen, C. T. Rettner, S. Raoux, G. W. Burr, S. H. Chen, R. M. Shelby, M. Salinga, W. P. Risk, T. D. Happ, G. M. McClelland, M. Breitwisch, A. Schrott, J. B. Philipp, M. H. Lee, R. Cheek, T. Nirschl, M. Lamorey, C. F. Chen, E. Joseph, S. Zaidi, B. Yee, H. L. Lung, R. Bergmann, and C. Lam. Ultra-thin phase-change bridge memory device using GeSb. In *International Electron Devices Meeting (IEDM 2006)*, pages 1–4, December 2006.

[31] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM Conference on Digital Libraries*, Berkeley, CA, USA, August 1999.

[32] Bogdan S. Chlebus and Dariusz R. Kowalski. Gossiping to reach consensus. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 220–229, Winnipeg, Manitoba, Canada, August 2002.

[33] Sung Jin Choi, Hee Yong, and Bo Kyoung Lee. An efficient dispersal and encryption scheme for secure distributed information storage. In *Computational Science - ICCS 2003*, volume 2660/2003 of *Lecure Notes in Computer Science*, pages 958–967. Springer Berlin, 2003.

[34] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John D. Kubiatowicz. Tiered fault tolerance for long-term integrity. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 2009)*, pages 267–282, San Francisco, CA, USA, February 2009.

[35] Michael Cieply. The afterlife is expensive for digital movies. *The New York Times*, December 2007.

[36] James Cipar, Mark D. Corner, and Emery D. Berger. TFS: A transparent file system for contributory storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 215–229, San Jose, CA, USA, February 2007.

[37] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009/2001, Berkeley, CA, USA, July 2000. Springer Berlin.

[38] Cleversafe. Technology white paper — highly secure, highly reliable, open source storage solution, June 2006.

[39] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–11, Baltimore, MD, November 2002.

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, Boston, MA, USA, second edition, 2001.

[41] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, CA, USA, December 2004.

[42] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshal, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2007)*, pages 205–220, Stevenson, WA, USA, October 2007.

[43] Ajay Dholakia, Evangelos Eleftheriou, Xiao-Yu Hu, Ilias Iliadis, Jai Menon, and KK Rao. Analysis of new intra-disk redudancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMetrics 2006)*, pages 373–374, Saint Malo, France, June 2006. ACM Press.

[44] T. Dierks and E. Rescorla. RFC 4346 - the transport layer security (TLS) protocol version 1.1, April 2006.

[45] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[46] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, USA, August 2004.

[47] John R. Douceur. The sybil attack. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7-8, 2002. Revised Papers*, volume 2429/2002 of *Lecure Notes in Computer Science*, pages 251–260, Cambridge, MA, USA, March 2002. Springer Berlin.

[48] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 617–624, Vienna, Austria, July 2002.

[49] Ann L. Chervenak Drapeau and Randy H. Katz. Striped tape arrays. Technical Report UCB/CSD-93-730, EECS Department, University of California, Berkeley, 1993.

[50] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 2009)*, pages 197–210, San Francisco, CA, USA, February 2009.

[51] Energy Information Administration. Monthly energy review January 2009. `http://www.eia.doe.gov/emeu/mer/contents.html`, January 2009.

[52] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the Fifth Annual ACM/IEEE International Conferece on Mobile Computing and Networking (MOBICOM 1999)*, pages 263–270, Seattle, WA, USA, August 1999.

[53] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72, Cape Cod, MA, May 1997.

[54] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and System (TPLS)*, 5(1):66–77, January 1983.

[55] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, February 2003.

[56] Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman. Optimizing power consumption in large scale storage systems. In *Proceedings of the 11th*

USENIX Workshop on Hot Topics in Operating Systems (HotOS 2007), San Diego, CA, USA, May 2007.

[57] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[58] Hector Garcia-Molina. Elections in a distributed computing systems. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[59] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, Bolton Landing (Lake George), New York, USA, October 2003. ACM Press.

[60] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, November 2000.

[61] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1998)*, pages 92–103, San Jose, CA, USA, October 1998.

[62] T.J. Giuli, Petros Maniatis, Mary Baker, David S. H. Rosenthal, and Mema Roussopolous. Attrition defenses for a peer-to-peer digital preservation system. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 163–178, Anaheim, CA, USA, April 2005.

[63] Henry M. Gladney and Raymond A. Lorie. Trustworthy 100-year digital objects: durable encoding for when it's too late to ask. *ACM Transactions on Information Systems (TOIS)*, 23(3):299–324, July 2005.

[64] Laurence Goasduff and Carina Swedemyr. Gartner says look beyond power issue as pressure mounts for 'greener' IT. Press release, November 2007.

[65] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Depenedable Systems and Networks (DSN 2004)*, Edinburgh, UK, June 2004.

[66] Kevin M. Greenan and Ethan L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT 2006)*, pages 178–187, Seoul, South Korea, October 2006. ACM Press.

[67] Kevin M. Greenan, Ethan L. Miller, and Thomas Schwarz, S. J. Optimizing Galois field arithmetic for diverse processor architectures. In *Proceedings of the 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2008)*, Baltimore, MD, USA, September 2008.

[68] Aloke Guha. Solving the energy crisis in the data center using COPAN systems' enhanced MAID storage platform. White Paper, December 2006.

[69] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005)*, pages 60–71, Madison, WI, USA, June 2005. IEEE Computer Society.

[70] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI 2005)*, pages 143–158, Boston, MA, USA, May 2005.

[71] Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7-8, 2002. Revised Papers*, volume 2429/2002 of *Lecure Notes in Computer Science*, pages 130–140, Cambridge, MA, USA, March 2002. Springer Berlin.

[72] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource discovery in distributed networks. In *Proceedings of the eighteenth annual ACM symposium on principles of distributed computing (PODC 1999)*, pages 229–237, Atlanta, GA, USA, May 1999.

[73] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 139–146, Portland, OR, USA, August 2007. ACM Press.

[74] Maurice P. Herlihy and J.D. Tygar. How to make replicated data secure. Technical Report CMU-CS-87-143, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, August 1987.

[75] Helen Heslop, Simon Davis, and Andrew Wilson. An approach to the preservation of digital records. National Archives of Australia, December 2002.

[76] Bo Hong, Feng Wang, Scott A. Brandt, Darrell D. E. Long, and Thomas J. E. Schwarz, S. J. Using MEMS-based storage in computer system — MEMS storage architectures. *ACM Transactions on Storage (TOS)*, 2(1):1–21, February 2006.

[77] R. J. Honicky and Ethan L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, USA, April 2004.

[78] James Hughes, Charles Milligan, and Jacques Debiez. High performance RAIT. In *Proceedings of the 19th IEEE/10th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, College Park, MD, USA, April 2003.

[79] Arun Iyengar, Robert Cahn, Juan Garay, and Charanjit Jutla. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC 1998)*, Vienna Austria and Budapest, Hungary, September 1998.

[80] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure charactersitics. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, pages 111–125, San Jose, CA, USA, February 2008.

[81] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 29–42, San Francisco, CA, USA, March 2003. USENIX Association.

[82] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technolgies (FAST 2004)*, San Francisco, CA, USA, March 2004. USENIX Association.

[83] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 223–236, Bolton Landing (Lake George), New York, USA, October 2003. ACM Press.

[84] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[85] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. Safestore: A durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 129–142, Santa Clara, CA, USA, June 2007.

[86] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, November 2000.

[87] Shay Kutten, David Peleg, and Uzi Vishkin. Deterministic resource discovery in distributed networks. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*, pages 77–83, Crete Island, Greece, July 2001.

[88] Leslie Lamport. Paxos made simple. *ACM SIGART News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[89] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and System (TPLS)*, 4(3):382–401, July 1982.

[90] Butler W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, January 1974.

[91] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1996)*, pages 94–92, Cambridge, MA, USA, October 1996.

[92] Andrew W. Leung and Ethan L. Miller. Scalable security for large, high performance storage systems. In *Proceedings of the 2nd International Workshop on Storage Security and Survivability (StorageSS 2006)*, Alexandria, VA, USA, October 2006.

[93] Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napier, and Indrajit Roy. BAR gossip. In *Proceedings of the 7th USENIX Symposium of Operating Systems Design and Implementation (OSDI 2006)*, pages 191–204, Seattle, WA, USA, November 2006.

[94] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 121–136, San Francisco, CA, USA, December 2004.

[95] Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard. A cooperative internet backup scheme. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, USA, June 2003.

[96] Witold Litwin, Rim Moussa, and Thomas J. E. Schwarz, S. J. LH*$_{RS}$ — a highly-available scalable distributed data structure. *ACM Transactions on Database Systems (TODS)*, 3(30):769–811, September 2005.

[97] Witold Litwin and Marie-Anna Neimat. High-availability LH* schemes with mirroring. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (IFCIS 1996)*, pages 196–205, Brussels, Belgium, June 1996.

[98] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH* — a scalable, distributed data stucture. *ACM Transactions on Database Systems (TODS)*, 21(4):480–525, December 1996.

[99] Witold Litwin and Tore Risch. LH*g: A high-availability scalable distributed data structure by record grouping. *IEEE Transactions of Knowledge and Data Engineering*, 14(4):923–927, July 2002.

[100] Witold Litwin and Thomas J. E. Schwarz, S. J. Algebraic signatures for scalable distributed data structures. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, pages 412–423, Boston, MA, USA, March 2004.

[101] Nancy A. Lynch. *Distributed Algorithms*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1st edition, April 1996.

[102] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA, December 2002.

[103] Jo Maitland. Hospital ditches EMC Centera for long-term archiving. *SearchStorage.com*, 2005.

[104] Petros Maniatis, Mema Roussopolous, T. J. Giuli, David S. H. Rosenthal, and Mary Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems (TOCS)*, 23(1):2–50, February 2005.

[105] Petros Maniatis, Mema Roussopolous, TJ Guili, David S. H. Rosenthal, Mary Baker, and Yanto Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 44–59, Bolton Landing (Lake George), New York, USA, October 2003.

[106] Catherine C. Marshall. Rethinking personal digital archives, part 1. Four challenges from the field. *D-Lib Magazine*, 14(3/4), March 2008.

[107] Catherine C. Marshall. Rethinking personal digital archives, part 2. Implications for services, applications and institutions. *D-Lib Magazine*, 14(3/4), March 2008.

[108] Shannon McCaffrey. Disk with data on 2.9M Georgians lost. *Associated Press*, April 2007.

[109] Carloline McCarthy. IRS tapes missing in kansas city. *CNET News.com*, 1/22/07, January 2007.

[110] Jai Menon, David A. Pease, Robert Rees, Linda Duyanovich, and Bruce Hillsberg. IBM storage tank – a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[111] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptography*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, London, UK, 1987. Springer-Verlag.

[112] Ethan L. Miller, Darrell D. E. Long, William E. Freeman, and Benjamin C. Reed. Strong security for network-attached storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technology (FAST 2002)*, Monterey, CA, USA, January 2002. USENIX Association.

[113] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP 2001)*, Lake Louise, Alberta, Canada, October 2001.

[114] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, pages 253–267, San Jose, CA, USA, February 2008.

[115] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 15–28, San Diego, CA, USA, December 2008.

[116] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[117] Paul Nowoczynski, Nathan Stone, Jason Sommerfield, Bryon Gill, and J. Ray Scott. Slash — the scalable lightweight archival storage hierarchy. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)*, pages 245–252, Monterey, CA, USA, April 2005.

[118] Panasas Inc. Tiered parity. `http://www.panasas.com/tiered_parity.html`, June 2008.

[119] Michael Peterson, Gary Zasman, Peter Mojica, and Jeff Porter. 100 year archive requirements survey. SNIA Data Management Forum, January 2007.

[120] Zachary N. J. Peterson, Randal Burns, Giuseppe Ateniese, and Stephen Bono. Design and implementation of verifiable audit trails for a versioning file system. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 93–106, San Jose, CA, USA, February 2007.

[121] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS 2004)*, pages 68–78, Saint Malo, France, July 2004.

[122] Eduardo Pinheiro, Ricardo Bianchini, and Cezary Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMetrics 2006)*, pages 15–26, Saint Malo, France, June 2006.

[123] Byron Pitts. Bank of america security lapse. *CBS News*, 2/25/05, February 2005.

[124] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Expierience*, 27(9):995–1012, September 1997.

[125] C. Greg Plaxton, Mitul Tiwari, and Praveen Yalagandula. Online aggregation over trees. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10, Long Beach, CA, March 2007.

[126] W. Curtis Preston and Gary Didio. Disk at the price of tape? An in-depth examination. Copan Systems white paper, 2004.

[127] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technology (FAST 2002)*, Monterey, CA, USA, January 2002. USENIX Association.

[128] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[129] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, April 1989.

[130] Tal Rabin. Robust sharing of secrets when the dealer is honest or cheating. *J. ACM*, 41(6):1089–1109, November 1994.

[131] Alan Radding. Storage gets a dose of medical data. *Storage Magazine*, July 2008.

[132] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM 2001)*, pages 161–172, San Diego, CA, USA, August 2001. ACM Press.

[133] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[134] Roger Reeves and Hans Bärfuss. PDF/A - a new standard for long-term archiving. `http://www.pdfa.org`, January 2007.

[135] Michael K. Reiter and Aviel D. Rubin. Anonymous web transaction with Crowds. *Communications of the ACM*, 42(2):32–48, February 1999.

[136] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, mangement, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, May 2003.

[137] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 1–14, San Francisco, CA, USA, March 2003. USENIX Association.

[138] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David F. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, June 2001.

[139] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, USA, January 2002.

[140] Ronald L. Rivest. Chaffing and winnowing: Confidentiality without encryption, April 1998.

[141] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP 1991)*, pages 1–15, Pacific Grove, CA, USA, October 1991.

[142] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[143] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles ACM symposium on operating systems principles*, pages 188–201, Banff, Alberta, Canada, October 2001.

[144] Yasushi Saito, Svend Frølund, Alistar C. Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 48–58, Boston, MA, USA, October 2004.

[145] Yasushi Saito, Christos Karamaolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA, December 2002. USENIX.

[146] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the FAST 2002 Conference on File and Storage Technology (FAST 2002)*, Monterey, CA, USA, January 2002.

[147] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 1–16, San Jose, CA, USA, February 2007.

[148] Thomas Schwarz, S. J. and Ethan L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, Lisbon, Portugal, July 2006. IEEE Computer Society.

[149] Thomas J. E. Schwarz, S. J., Qin Xin, Ethan L. Miller, and Darrell D. E. Long. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, Volendam, Netherlands, October 2004.

[150] Secretary of Comerce. Entity authentication using public key cryptography (FIPS PUB 197), February 1997.

[151] Secretary of Comerce. Digital signature standard (DSS) (FIPS PUB 186-2), January 2000.

[152] Secretary of Comerce. Advanced encryption standard (AES) (FIPS PUB 197), November 2001.

[153] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[154] Aameek Singh, Madhukar Korupolu, and Kaladhar Voruganti. Zodiac: Efficient impact analysis for storage area networks. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 2005)*, pages 73–86, San Francisco, CA, USA, December 2005.

[155] Small Form Factors Committee SFF-8035. ATA SMART feature set commands. `http://www.t13.org`.

[156] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 191–202, Berkeley, CA, USA, 1988.

[157] Douglas R. Stinson. *Cryptography: Theory and Practive*. The CRC Press Series on Discrete Mathematics and its Applications. Chapman and Hall (CRC), Boca Raton, FL, USA, first edition, 1995.

[158] Douglas R. Stinson. *Cryptography: Theory and Practive*. The CRC Press Series on Discrete Mathematics and its Applications. Chapman and Hall (CRC), Boca Raton, FL, USA, second edition, 2002.

[159] Ian Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM 2001)*, pages 149–160, San Diego, CA, USA, August 2001. ACM Press.

[160] Michael Stonebraker and Gerhard A. Schloss. Distributed RAID – A new multiple copy algorithm. In *Proceedings of the Sixth International Conference on Data Engineering (ICDE '90)*, pages 430–437, Los Angeles, CA, USA, 1990.

[161] Mark W. Storer, Kevin Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th International Workshop on Storage Security and Survivability (StorageSS 2008)*, pages 1–10, Fairfax, VA, USA, October 2008.

[162] Mark W. Storer, Kevin Greenan, and Ethan L. Miller. Long-term threats to secure

archives. In *Proceedings of the 2nd International Workshop on Storage Security and Survivability (StorageSS 2006)*, Alexandria, VA, USA, October 2006.

[163] Mark W. Storer, Kevin Greenan, Ethan L. Miller, and Carlos Maltzahn. POTSHARDS: Storing data for the long-term without encryption. In *Proceedings of the 3rd International IEEE Security in Storage Workshop (SISW 2005)*, San Francisco, CA, USA, December 2005.

[164] Mark W. Storer, Kevin M. Greenan, Ian F. Adams, Ethan L. Miller, Darrell D. E. Long, and Kaladhar Voruganti. Logan: Automatic management for evolvable, large-scale, archival storage. In *Proceedings of the 3rd International Petascale Data Storage Workshop (PDSW '08)*, Austin, TX, USA, November 2008.

[165] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. POTSHARDS: Secure long-term storage without encryption. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 143–156, Santa Clara, CA, USA, June 2007.

[166] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Secure, archival storage with POTSHARDS. In *Works in Progress of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, San Jose, CA, USA, February 2007.

[167] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Energy-efficient archival storage with disk instead of tape. *;login:*, 33(3):15–21, June 2008.

[168] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, pages 1–16, San Jose, CA, USA, February 2008.

[169] Arun Subbiah and Douglas M. Blough. An approach for fault tolerant and secure data storage in collaborative work environments. In *Proceedings of the First International Workshop on Storage Security and Survivability*, pages 84–93, Fairfax, VA, USA, November 2005. ACM.

[170] Sun Microsystems. Solaris ZFS and Red Hat Enterprise Linux EXT3 file system performance. http://www.sun.com/software/whitepapers/solaris10/zfs_linux.pdf, June 2007.

[171] Sun Microsystems. Sun StorageTek 5800 system architecture. http://www.sun.com/storagetek/disk_systems/enterprise/5800, December 2007.

[172] Takaya Tanabe, Makoto Takayanagi, Hidetoshi Tatemiti, Tetsuya Ura, and Manabu Yamamoto. Redundant optical storage system using DVD-RAM library. In *Proceedings of the 16th IEEE Symposium on Mass Storage Systems (MSS 1999)*, pages 80–87, San Diego, CA, USA, March 1999.

[173] S. Tehrani, J.M. Slaughter, E. Chen, M. Durlam, J. Shi, and M. DeHerren. Progress and outlook for MRAM technology. *IEEE Transactions on Magnetics*, 35(5):2814–2819, September 1999.

[174] The Green Grid. The green grid opportunity, decreasing datacenter and other IT energy usage patterns, February 2007.

[175] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 224–237, Saint Malo, France, October 1997.

[176] Carmela Troncoso, Danny De Cock, and Bart Preneel. Improving secure long-term archival of digitally signed documents. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS 2008)*, pages 27–36, Fairfax, VA, USA, October 2008.

[177] Emmanouel A. Varvarigos and Dimitri P. Bertsekas. Performance of hypercube routing schemes with or without buffering. *IEEE Transactions on Networking*, 2(3):299–311, June 1994.

[178] Sudarshan Vasudevan, Brian DeCleene, Neil Immerman, Jim Kurose, and Don Towsley. Leader election algorithms for wireless ad hoc networks. In *Proceedings of the IEEE DARPA Information Survivability Conference and Exposition (DISCEX 2003)*, volume 1, pages 261–272, Washington, DC, USA, April 2003.

[179] Surdarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP 2004)*, pages 350–360, Berlin, Germany, October 2004.

[180] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, pages 59–72, Denver, CO, August 2000.

[181] Chip Walter. Kryder's law. *Scientific American*, July 2005.

[182] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, Alexandria, VA, USA, October 2006.

[183] Julie Ward, Michael O'Sullivan, Troy Shahoumian, and John Wilkes. Appia: automatic stoarage area network fabric design. In *Proceedings of the FAST 2002 Conference on File and Storage Technology (FAST 2002)*, pages 203–217, Monterey, CA, USA, January 2002.

[184] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[185] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: Exploting a secure log for wide-area distributed storage. In *Proceedings of the 2nd European Conference on Computer Systems (Eurosys 2007)*, pages 371–384, Lisbon, Portugal, March 2007.

[186] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, and Ken Birman. Smoke and mirrors: Reflecting files at a geographically remote location without loss of performance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 2009)*, pages 211–224, San Francisco, CA, USA, February 2009.

[187] Hakim Weatherspoon, Hugo Miranda, Konrad Iwanicki, Ali Ghodsi, and Yann Busnel.

Gossiping over storage systems is practical. *ACM Operating Systems Review (OSR)*, 41(5):75–81, October 2007.

[188] Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang, Peter Reiher, and Geoff Kuenning. PARAID: A gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 245–260, San Jose, CA, USA, February 2007.

[189] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium of Operating Systems Design and Implementation (OSDI 2006)*, pages 307–320, Seattle, WA, USA, November 2006.

[190] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, pages 17–33, San Jose, CA, USA, February 2008.

[191] Western Digital. WD Caviar GP 1 TB SATA hard drives. `http://www.westerndigital.com/en/library/sata/2879-701229.pdf`, August 2007.

[192] W. W. Wilcke, R. B. Garner, C. Fleiner, R. F. Freitas, R. A. Golding, J. S. Glider, D. R. Kenchammana-Hosekote, J. L. Hafner, K. M. Mohiuddin, KK Rao, R. A. Becker-Szendy, T. M. Wong, O. A. Zaki, M. Hernandez, K. R. Fernandez, H. Huels, H. Lenk, K. Smolin, M. Ries, C. Goettert, T. Picunko, B. J. Rubin, H. Kahn, , and T. Loo. IBM intelligent bricks project – petabytes and beyond. *IBM Journal of Research and Development*, 50(2/3):181–197, March 2006.

[193] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe - the least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS 2008)*, pages 21–26, Fairfax, VA, USA, October 2008.

[194] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullican. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, February 1996.

[195] Theodore M. Wong, Chenxi Wang, and Jeannette M. Wing. Verifiable secret redistribution for archive systems. In *Proceedings of the First International IEEE Security in Storage Workshop*, Greenbelt, MD, USA, December 2002.

[196] Theodore M. Wong, Chenxi Wang, and Jeannette M. Wing. Verifiable secret redistribution for threshold sharing schemes. Technical Report CMU-CS-02-114-R, Carnegie Mellon University, October 2002.

[197] Jay J. Wylie, Michael W Bigrigg, John D. Strunk, Gregory R. Ganger, Hans Kiliççöte, and Pradeep K. Khosla. Survivable information storage systems. *Computer*, 33(8):61–68, August 2000.

[198] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and responses to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, pages 223–234, Alexandria, VA, USA, November 2005.

[199] Praveen Yalagandula and Mike Dahlin. A scalable distributed information management system. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM 2004)*, pages 379–390, Portland, OR, USA, August 2004. ACM Press.

[200] Praveen Yalagandula and Mike Dahlin. Shruti: A self-tuning hierarchical aggregation system. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 141–150, Boston, MA, USA, July 2007. IEEE Computer Society.

[201] Pete Yost. DOJ: Misplaced laptops still a problem. *Wired News*, February 2007.

[202] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, Tokyo, Japan, April 2005.

[203] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 77–92, San Jose, CA, USA, February 2007.

[204] Lingfang Zeng, Dan Feng, Fang Wang, Ke Zhou, and Peng Xia. Hybrid RAID-tape-library storage system for backup. In *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS 2005)*, pages 31–36, Xi'an, China, December 2005. IEEE Computer Society.

[205] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, and Chao Jin. BitVault: A highly reliable distributed data retention platform. Technical Report MSR-TR-2005-179, Microsoft Research Asia, December 2005.

[206] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, and Chao Jin. BitVault: A highly reliable distributed data retention platform. *ACM SIGOPS Operating Systems Review*, 41(2):27–36, April 2007.

[207] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service development. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[208] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the Twentieth ACM Symposium on Operator Systems Principles (SOSP 2005)*, pages 177–190, Brighton, United Kingdom, October 2005.