POTSHARDS: Secure Long-Term Archival Storage Without Encryption

Technical Report UCSC-SSRC-06-03

Mark W. Storer
mstorer@cs.ucsc.edu

Kevin Greenan
kmgreen@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Kaladhar Voruganti
kaladhar@us.ibm.com

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

September 27, 2006

POTSHARDS: Secure Long-Term Archival Storage Without Encryption

Mark W. Storer
mstorer@cs.ucsc.edu

Kevin Greenan
kmgreen@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Kaladhar Voruganti
kaladhar@us.ibm.com

Abstract

Modern archival storage systems either store data in the clear, ignoring security, or rely on keyed encryption to ensure privacy. However, the use of encryption is a major concern when data must be stored an indefinite period of time—key management becomes increasingly difficult as file lifetimes increase, and data loss becomes increasingly likely because keys are a single point of failure and losing a key is comparable to data deletion. Moreover, traditional systems are subject to the obsolescence of encryption algorithms themselves, which can expose petabytes of data the instant a cryptographic algorithm is broken.

To address these concerns, we developed POTSHARDS, an archival storage system that addresses the long-term security needs of data with very long lifetimes without the use of encryption. POTSHARDS separates security and redundancy by utilizing two levels of secret splitting in a way that allows the original data to be reconstructed from the stored pieces. However, the data structures used in POTSHARDS are also designed in such a way that an unauthorized user attempting to collect sufficient shares to reconstruct any data will not go unnoticed because it is very difficult to launch a targeted attack on the system, even with unrestricted access to a few archives. Since POTSHARDS provides secure storage for arbitrarily long periods of time, its data structures include built-in support for consistency checking and data migration. An evaluation of our POTSHARDS implementation shows that it stores and retrieves data at 2.5–5 MB/s, demonstrates its ability to recover user data given all of the pieces a user has stored across the archives, and shows its ability to recover from the loss of an entire archive.

1 Introduction

The capabilities of current archival storage facilities lag behind the flood of data that must be archived in digital form. Recent legislation, such as Sarbanes-Oxley and HIPPA, have placed strict demands on the preservation and retrieval properties of storage systems. Beyond the relatively short (for archival purposes) lifetimes enforced by such legislation, there is information that must last at least as long as the owner's lifetime. For example, many medical records, legal documents, corporate records, and historical data must be preserved indefinitely with a high degree of security.

To address the many security requirements for long-term archival storage, we designed and implemented POTSHARDS (Protection Over Time, Securely Harboring And Reliably Distributing Stuff). The key ideas in POTSHARDS are the use of *secret splitting* techniques to distribute data across independent authentication domains and the use of multiple levels of secret splitting and approximate pointers to hide the relationship between the resulting pieces. By providing data secrecy without the use of encryption, POTSHARDS is able to move security from encryption to the more flexible and secure authentication realm; unlike encryption, authentication need not be done by computer, and authentication schemes can be easily changed in response to new vulnerabilities.

Archival storage exhibits a write-once, read-maybe usage model, in contrast to more active storage that is read more frequently than written. This access model, in combination with potentially indefinite data lifetimes, presents many novel storage problems [2]. Long-term archives will experience many media and hardware changes due to failure and technological advances, but data must be preserved across this evolution. Solutions that suffice for short-term applications begin to degrade as timeframes move from months to decades. For example, the management of cryptographic keys becomes difficult as data may experience many key rotations and cryptosystem migrations over the course of several decades.

The long-term use of encryption in archives is a major concern for a number of reasons. First, there is a high chance that, over the course of many years, users will frequently lose their encryption keys, and key loss is often equivalent to data deletion. Second, encryption keys can be stolen, potentially compromising large quantities of archived data. Both of these issues affect all of the data that was encrypted using a particular key. Third, encryption standards may change over the course of many years, both because of the advances in computational ability to "brute-force" an algorithm and because algorithms can be compromised by cryptanalysis. Fourth, long-lived data may require many key rotations over its lifetime. Addressing the issue often raises the recurring problem of migrating large amounts of encrypted data to new encryption keys or even new encryption algorithms. While re-encryption may be straightforward for small amounts of data, migrating petabytes of data in response to a compromised encryption algorithm is very difficult and either requires active user participation or allowing the archive to know the key used, neither of which is attractive for long-term secure storage. These and similar issues will likely be encountered over the long data lifetimes found in the archival storage realm.

The design of POTSHARDS includes three novel aspects that make it well-suited for usage as a secure archive. First, POTSHARDS uses secret splitting and approximate pointers as a way to move security from encryption to authentication and to avoid reliance on encryption algorithms that may be compromised at some point in the future—unlike encryption, secret splitting provides information-theoretic security. Second, each user maintains a separate, recoverable index over her data, so a compromised index does not affect the other users and a lost index is not equivalent to data deletion. More importantly, in the event that a user loses her index, both the index and the data itself can be securely reconstructed from the user's shares stored across multiple archives. However, this reconstruction requires an exponential number of shares because only the data owner knows the exact relationship between the shares. An attacker must use approximate pointers to identify sets of shares to retrieve and illicitly assemble all of them without any archive noticing the intrusion. Third, POTSHARDS uses a write-once RAID technique across multiple archives to provide redundancy, data migration, recoverability and integrity checking in a distributed environment.

We implemented POTSHARDS in 15,000 lines of Java, and conducted several experiments to test its performance and resistance to failure. POTSHARDS can read and write data at 2.5–5 MB/s on commodity hardware, and survives the failure of an entire archive with no data loss and little effect seen by users. In addition, we demonstrated the ability to reconstruct a user's data from all of the user's stored shares. These experiments demonstrate the system's suitability to the unique usage model of long-term archival storage.

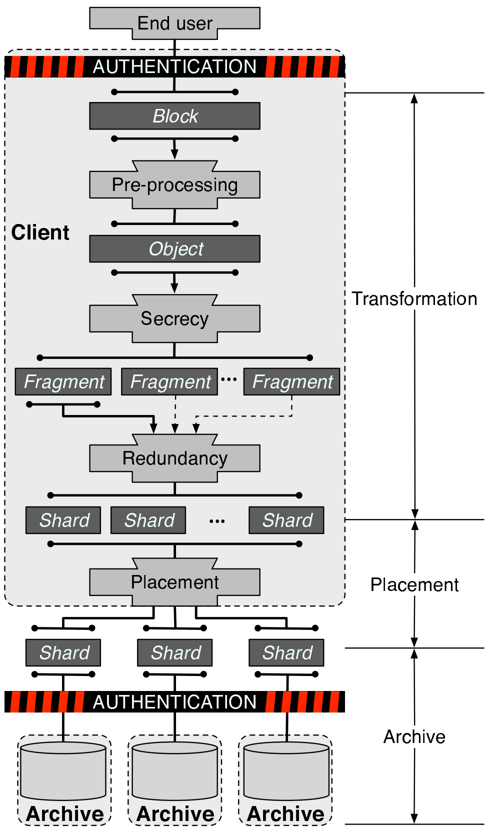


Figure 1: *Overview of the POTSHARDS archival storage system. The arrows in this diagram point down, corresponding to data being ingested. For data extraction, the arrows (and data flow) point up towards the user.*

2 Architecture Overview

POTSHARDS was designed to provide long-term secure archival storage that avoids the use of encryption. Thus, it is important to describe the long-term threats that POTSHARDS attempts to guard against. POTSHARDS uses three techniques to counter these threats: secret splitting, a technique that, unlike encryption, provides information-theoretically *provable* security; approximate pointers that hide direct relationship between pieces of stored data; and separation of archives into distinct authorization domains.

The POTSHARDS model assumes a relatively stable environment in which the barrier to entry for a new archive would be relatively high because it would need to demonstrate that it could provide a certain level of service. POTSHARDS is not designed as a loosely federated peer-to-peer service; rather, each archive should have an incentive to take an active role in preserving its own stability and security while actively questioning the stability and security of the other archives. However, since POTSHARDS must survive for decades, it is a given that the overall system will undergo constant change as technology improves. The design of POTSHARDS, shown in Figure 1 makes it easier to accommodate these incremental changes in two key ways. First, the system is designed as a series of modules. A modular design allows each component to be optimized for a specific task. Second, communication between components is performed through well-defined interfaces using a small number of request and response message types.

The remainder of this section will discuss the threat model that long-term secure archives such as POTSHARDS must face (Section 2.1), and the data organization that POTSHARDS uses to combat these threats (Section 2.2). The software structure and organization of the system itself is discussed in Section 3.

2.1 Threat Model

In order to properly design a system for secure archival storage, it is necessary to identify the unique threats to such long-term storage systems. Some of these threats are variations on common concerns that take on new meaning in the area of archival storage, while others are new threats that typically do not affect traditional storage systems. Adequate understanding of these threats is essential to effectively devise new policies and mechanisms to guard against them. We will focus on threats to the security of the long-term archive; more conventional threats are described elsewhere [2].

Secrecy in long-term storage presents a complex challenge. 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. Thus, secrecy techniques must adequately balance the need for secrecy and the unrecoverable nature of the data. Additionally, although the usage model of storage is write-once, read-maybe, users must still be able to find the data they stored should they desire to read it. If a user is unable to locate and retrieve his data from an archival system, the availability aspect of security has been violated.

Authentication and data availability in long-term secure storage also presents problems not typically encountered in shorter-term storage systems. If the storage system plans on providing file secrecy as part of security, users must be able to authenticate themselves to the system as a first step in authorization. A challenge in long-term secure storage is that the user primarily attached to the data may no longer be available, and the credentials may be lost as well (*i. e.*, perhaps the user has died). Archival storage systems must be able to produce data if a party provides sufficient authorization; "we lost the key" is not an excuse for denial of service.

Intrusion detection is more challenging in archival systems because the long data lifetimes that such systems must support provide attackers with much longer windows during which they can attempt to compromise a system's security. Traditional intrusion detection includes techniques that compare audit data and network activity to a database of known attacks. However, an attack that is methodical enough to make only the slightest of changes at any one time and space each step far enough apart would be difficult to detect by traditional signature matching algorithms. For example, an attack that required an intruder to break into a system to steal twenty pieces of data would likely be caught if the attack occurred during a two day period, but would be almost certainly be missed if the attack took place over two decades.

Maintaining integrity is likewise more difficult because of the relatively short lifetime and limited reliability of traditional storage components, causing problems with data degradation over the data's lifetime, which greatly exceeds individual component lifetime. Integrity guarantees involve two distinct issues. The first is the internal integrity problem of data within an archive becoming corrupted by media failures and silent read errors. The second, found in distributed storage systems, is the external integrity challenge. Each archive must have a method of ensuring that the other archives in the distributed system are behaving properly. This problem has been addressed for public data [9, 17], but not for secure storage. Due to the long lifetimes of archival data, long-term storage systems will witness events that require data to be moved between archives. The reasons for this change could include the inevitable failure or obsolescence of hardware, system updates and possibly even data movement as a security factor. A long-term system must thus be immune to the failure or reliance of any given component.

2.2 Data Entities

Data inserted into POTSHARDS goes through multiple layers of a software stack to distribute information in such a way that an attacker cannot easily reconstruct the data, as shown in Figure 1. Data at the top level consists of *files*, which are the entities that clients submit to the system for preservation. Files are broken into fixed-size blocks, which are converted into b-byte *objects* by appending the secure hash of the underlying block and the ID of the object. Objects are then split into *fragments* using an algo-

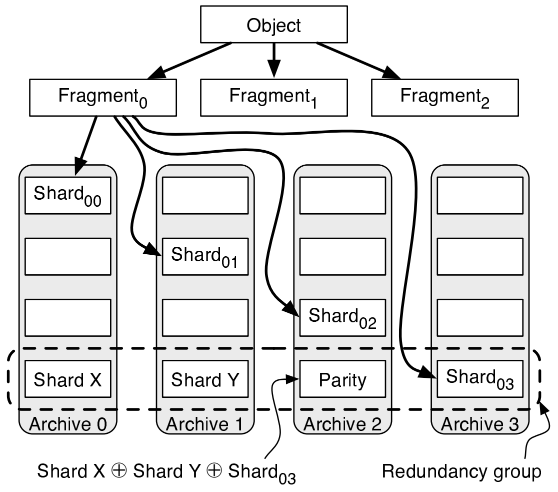


Figure 2: *The relationship between objects, fragments, and shards in POTSHARDS. Archive-level redundancy, described in Section 3.3.3, is done across unrelated shards.*

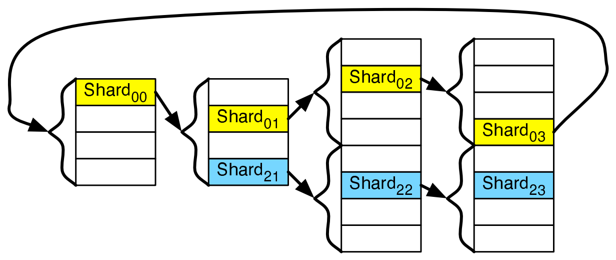


Figure 3: *Approximate pointers and shard relationships. In this diagram, each approximate pointer points to $R = 4$ possible shards. Fragment 0 can be reconstructed by following the links for shard$_{0X}$. If an intruder mistakenly picks shard$_{21}$, he will not discover his error until he has retrieved the entire chain and attempted to verify the reassembled fragment.*

rithm optimized for secrecy with little regard for redundancy. Currently, POTSHARDS uses an XOR-based secret splitting algorithm for this stage, which produces n fragments, each of which is b bytes long. In addition to the data needed to reconstruct the object, each fragment contains information used to reconstruct the fragment from the *shards* into which the fragment is broken. The relationship between objects, fragments, and shards is shown in Figure 2.

Shards are the basic entity stored in the archives, and, in this version of POTSHARDS, are generated from fragments using an m/n secret splitting algorithm [22, 28]. In secret splitting, a secret is distributed by splitting it into a set number n of shares such that no group of $k < m$ shares reveals any information about the secret; this approach is called an (m,n) threshold scheme. In such a scheme, any m of the n shares can be combined to reconstruct the secret, but there are information-theoretic proofs that combining fewer than m shares reveals **no** information. Thus, splitting fragments using an m/n scheme results in n shards that each contain b bytes of data. Shards contain no information about the fragments that make them up; as a result, an attacker who compromises an archive cannot gain information about the data being stored on the archive other than its source, which can, if desired, be further hidden using onion routing [10] or similar techniques. Note that, because of the two-level secret splitting, the overall storage requirements for the system are relatively high; a 2-way XOR split followed by a $2/3$ secret split would increase storage requirements by a factor of six. However, users can easily submit compressed archival data [34] to be stored, reducing the amount of storage needed; compressed data is handled just like any other type of data.

The only information a shard contains to link it to other shards of the same fragment is an *approximate pointer*. Each shard has an approximate pointer to the next shard in the fragment, and the last shard points back to the first, completing the cycle, as shown in Figure 3. In contrast to traditional pointers that contain the exact identifier of the object pointed to, an approximate pointer points to a *range* of potential identifiers. This can be implemented in two ways: randomly picking a value within $R/2$ above or below the next shard's identifier, or masking off the low-order r bits ($R = 2^r$) of the next shard's identifier, hiding the true value. Currently, POTSHARDS uses the latter approach; we are investigating the tradeoffs between the two approaches. When shards are created, the *exact* names of the shards are returned to the user. Typically, a user maintains the relationship between shards, fragments, objects, and files in an index to allow for fast retrieval, as described in Section 3.2; however, these exact pointers are *not* stored in the shards themselves, so they are not available to someone attacking the archives.

The use of approximate pointers provides a great deal of security by preventing an intruder who compromises an archive from knowing exactly which shards to steal from other archives. An intruder would have to steal *all* of the shards the approximate pointer could refer to, and would have to steal all of the shards they refer to, and so on. All of this would have to bypass the authentication mechanisms of each archive, and archives would be able to identify the access pattern of a thief, who would have to steal consecutively-numbered shards from an archive. Since partially reconstructed fragments cannot be verified, the intruder might have to steal *all* of the potential shards to ensure that he was able to reconstruct the fragment. For example, if an approximate pointer points to R shards and a fragment is split using m/n secret splitting, an intruder would have to steal, on average, $R^{m-1}/2$ shards to decode the fragment.

A legitimate user who has access to all of his shards, on the other hand can easily rebuild the fragments and, from them, the objects and files they comprise. Recovery of the user's data is based on approximate pointers located within the shards, as shown in Figure 4. Once a user gains all of the necessary shards, there are two brute-force approaches to regenerating the fragments encoded into shards using m/n coding. First, a user could try every possible chain of length m, rebuilding the fragment and attempting to verify it. Second, a user could narrow the list of possible chains by only attempting to verify chains of length n that represented cycles, a process we call the *ring heuristic*. The Shamir secret splitting algorithm is computationally expensive, so combining a set of shards that do not produce a valid fragment is expensive. The ring heuristic reduces the number of failed reconstruction attempts in two ways. First, the number of cycles of length n is lower than the number of paths of length m since many paths of length n do not make cycles. Second, reconstruction using the Shamir secret splitting algorithm requires that the shares be properly ordered and positioned within the share list. Though the shard ID provides a natural ordering for shards, it does not assist with positioning. For example, suppose the shards were produced with a 3 of 5 split. A chain of three shards, $\langle s_1, s_2, s_3 \rangle$, would potentially need to be submitted to the secret splitting algorithm three times to test each possible index: $\langle s_1, s_2, s_3, \phi, \phi \rangle$, $\langle \phi, s_1, s_2, s_3, \phi \rangle$, and $\langle \phi, \phi, s_1, s_2, s_3 \rangle$. As Figure 4 illustrates, fragments include a hash which is used to confirm successful reconstruction. Fragments also include the identifier for the object from which they are derived, making the combination of fragments into objects a straightforward process. An evaluation of the two approaches to select chains to verify is discussed further in Section 4.2.

All data entities are given 128-bit identifiers; objects, fragments and shards all have unique names within the system. The first 40 bits of the name uniquely identify the client in the same manner as a bank account is identified by an account number. The remaining 88 bits are used to identify the data entity. Object IDs and fragment IDs do not play a role in the security of the data, so their names can be generated simply. In contrast, the time to recover objects from a set of shards is directly related to the density of the shards' names—higher densities make recovery slower and allow shards to "hide" amongst more shards. Thus, shards' IDs must be chosen with greater care to ensure a high density of names to provide sufficient security.

In addition to uniquely identifying data entities within the system, IDs play an important role in the secret-splitting algorithms used in POTSHARDS. For secret-splitting techniques that rely on linear interpolation [28], the order of the secret shares is an input to the reconstruction algorithm. Thus, knowing the order of the shards in a ring can greatly reduce the time taken to reconstruct the secret. This is currently done by ensuring that the shards that make up a fragment have identifiers that follow the shards' input order to the reconstruction algorithm.

3 System Design

This section describes the software structure of POTSHARDS and the archives it uses. Both the POTSHARDS and archive sections discuss the components that make up the system and their relationships to one another, and the flow of information through the components. The archive description is separate from the POTSHARDS design because the internal archive design is largely orthogonal from the rest of POTSHARDS.

3.1 POTSHARDS Components and Data Flow

As Figure 1 shows, POTSHARDS consists of three primary layers: transformation, placement and archive. Layers communicate with one another through a request and response message protocol. Request messages travel down the stack from the client application to the archives and response messages travel back up the stack. Requests start at the top of the stack with the client application. Below this is the transformation layer, which takes those blocks of data and, utilizing secret splitting, provides data secrecy and user-level redundancy. The next layer is responsible for placement, accepting the data entities to store in the system and distributing them to the archive layer.

Each archive in POTSHARDS can exist within its own security domain with its own authentication scheme. Though POTSHARDS is distributed, it is different from many peer-to-peer systems in that archives do not join and leave the system frequently. Furthermore, each archive is expected to ensure its own stability and integrity while actively auditing the stability and integrity of its partner archives. Archive internals are further described in Section 3.3.

The modular design and communications model allow a large degree of flexibility in where the layers reside and even allow multiple users to share a single instance of a layer. One model that provides a high degree of security is to place the client application, transformation layer and placement layer on the user's local computer. In this manner, the user's unsecured data is never transmitted over an open communications channel. Alternatively,

Module	*Input*	*Output*
Pre-processing	block	object
Secrecy split	object	set of fragments
Redundancy split	fragment	set of shards

Table 1: *Transformation layer modules and their inputs and outputs.*

network communications can travel over SSL-encrypted links, providing strong security for data in transit.

The user communicates with POTSHARDS through a client application, which has three primary tasks. In the first task, *file ingestion*, the client breaks the file into fixed-sized blocks, adding padding on the end of the last block if necessary. The client then submits these blocks to the system as a storage request. If successful a storage response contains the information needed to map shards to blocks. The second task of the client is to request blocks from the system. Extraction is based on the block to shard mapping contained within an index maintained by the client. Requests for shards travel down the stack along with information needed to reconstruct the block. As the response travels back up the stack, the block is reconstructed. The third task of the client is to manage the user's index to provide the shard identifiers used to retrieve blocks from the system. This task, described in detail in Section 3.2, includes constructing index pages, submitting pages to the system for storage and requesting index pages from the system. The first two tasks are required tasks for POTSHARDS; however, the third task is optional—the user can choose to maintain the index locally and not store it in the archive.

The **transformation layer** is responsible for encoding data during ingestion and reconstruction during extraction. In keeping with the specialized components design of POTSHARDS, data transformation is accomplished using three distinct phases: pre-encoding, secrecy encoding, and user-level redundancy encoding. The inputs and outputs to each layer can be seen in Table 1.

The first phase, pre-encoding, produces objects from files via blocks. As Figure 4 illustrates, an object in POTSHARDS contains a hash over the object ID and object payload. During extraction, this hash can be used to confirm the successful reconstruction of an object.

The second phase is tuned for secrecy and involves an n/n secret split using an XOR-based algorithm. It can be shown that splitting using XOR, in addition to being a relatively fast secret splitting technique, results in provably secure data secrecy. This transformation phase takes objects as input and produces a set of fragments. As Figure 4 shows, fragments also contain a hash over their contents that can be used to confirm a successful reconstruction.

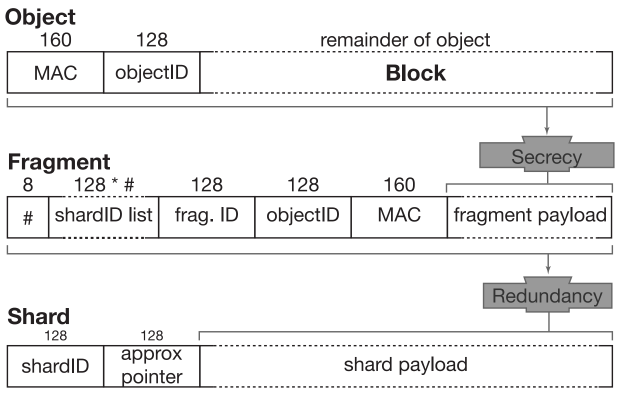


Figure 4: *Data entities in POTSHARDS. Size, in bits, is indicated above each field. Note that entities are not shown to scale relative to one another. # is the number of shards that the fragment produces*

The fragment also contains metadata that can be used to identify the shards that it produces as well as the object that it was derived from.

The third phase produces a set of shards and is based on Shamir's m/n linear interpolation secret splitting algorithm [28]. This layer is tuned for user-level redundancy, and provides security over the fragment's metadata. User-level redundancy is used to provide availability in the event that an archive is unavailable during extraction. The only metadata that shards contain are an identifier and an approximate pointer to another shard. Thus, a lone shard does not reveal any information about the object that it is used to rebuild, the exact ID of the next shard or even the location of the next shard.

The **placement layer** receives sets of shards from the transformation layer and is responsible for two tasks. The first is reducing the risk of information leakage. It does this by comparing the parameters used to generate shards, the placement policy and the number of archives. The second task, if the constraints are met, is to assign shards to archives. The assignment ensures that no two shards from a single fragment are placed on any given archive.

3.2 Archive Index

While the archival model of storage is write-once, read-maybe, users must still be able to find the data they stored should they desire to read it. Thus, users who store data in POTSHARDS should keep an index that stores the mapping between files and shards, allowing the user to locate the shards that are needed to rebuild (via fragments, objects, and blocks) a given file. The user's index is similar to an encryption key in that it contains the information needed to reconstruct the user's data. However, unlike data secured by encryption, a users's data can be recovered *without* an index.

Of course, this index is at risk of loss; while data can be retrieved without it, its presence greatly simplifies data retrieval. Thus, users that store their data in POTSHARDS may also store their indexes in the system. If this is done, the user need only rebuild the index using the algorithm described in Section 2.2; the rebuilt index can then be used to find the shards needed to rebuild any object. This approach has two advantages. First, since each user maintains his own index, if the index for one user is compromised, it does not affect the security of other users' data. Second, the index for one user can be recovered with no effect on other users.

Because the index is recoverable from the data itself, it is different from using encryption in two important ways. First, the user's index is not a single point of failure like an encryption key. 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 reconstruction of the data. If the data was encrypted, the files without the encryption key are effectively inaccessible. However, POTSHARDS can rebuild the data without the index or any other user input if all of the shards are available.

The index for each user is stored as a linked list of index pages with new pages inserted at the head of the list. Since the 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. Included in the new page is a list of shards corresponding to the previous head of the list. This new page is then submitted to the system and the shard list returned is maintained as the 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. In the event that a user loses her index, she can authenticate to the archives (perhaps complying with more stringent authentication policies) and retrieve the shards that belong to her. Once the client's shards have been collected, the approximate pointers can be used as hints in the combinatoric problem of combining shards, as described in Section 2.2.

3.3 Archive Design

The architecture of POTSHARDS demands an archive model that preserves the secrecy provided by the other components of the system while maintaining the goal of reliable, long-term storage. The system makes both reliability and security guarantees by arranging each archive into a separate security-failure domain and intelligently placing the shards across each domain. In addition, the archives themselves hold no information about fragment and object reconstruction, so a full compromise of a single archive gives an adversary very little, if any, information that can be used to recover user data. Absent such precautions, the archive model would likely weaken the strong security properties provided by the other system components.

3.3.1 Components

Archives in POTSHARDS are oblivious to the existence of files, objects and fragments; their only job is to reliably store shards for an extended period of time without revealing any information about client data. Reliability is achieved by requiring all archives to agree on RAID-based methods to ensure whole archive reconstruction in the presence of failure. In the absence of RAID techniques across the archives, the procedure of archive reconstruction would involve requesting and scanning user indices, which would compromise security and violate the POTSHARDS security property of having only a client know the explicit relationships between its shards.

Redundancy synchronization is controlled by a set of replicated *recovery managers*. This coordination allows the archives to form coherent, fault-tolerant *redundancy groups* without disclosing any information about the origin of the individual shards. Each recovery manager holds the system-wide redundancy group indices. Due to the importance of redundancy information, the recovery manager is replicated, giving each archive its own recovery manager. Updates such as archive and storage additions are assumed to occur occasionally, so maintaining consistency across the replicated managers requires a manager to broadcast any changes in its index to all other managers on the system's archives.

An archive joins the POTSHARDS system by first formatting all of its available storage into equal-sized *shard groups*. Each shard group contains a hash of its contents, an ID header and an array of equal-sized, write-once *shard slots*. The hash is used to perform intra-archive integrity checks; the ID header contains a map of the shards stored in the shard group; and each fixed-size shard slot contains shard data. Next, the archive will advertise all newly created shard groups to a recovery manager, which will assign the shard groups to equal-sized *virtual disks*. The virtual disks are used to create redundancy groups and allow for maximum space utilization across heterogeneous archives.

A redundancy group is a set of n virtual disks, partitioned into p parity disks and $n-p$ data disks. Each redundancy group is constructed with a particular RAID al-

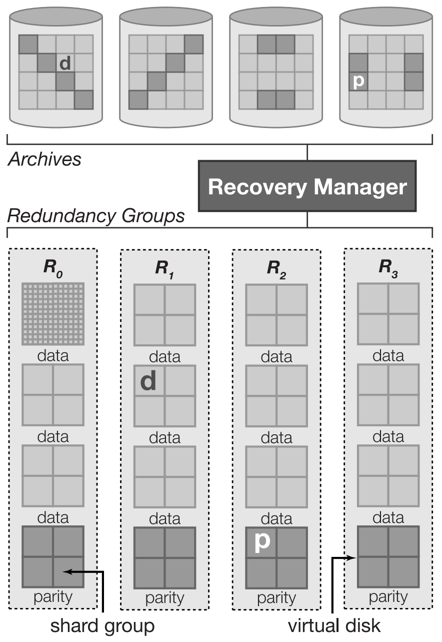


Figure 5: *Shard groups arranged into virtual disks and redundancy groups. The first virtual disk of group R_0 shows that shard groups are made of fixed sized shard slots.*

gorithm, which has the ability to tolerate any *p* virtual disk failures. To ensure a prescribed level of archive fault tolerance, a set of constraints based on first-fit bin-packing, *n*, *p* and the RAID algorithm are used to assign virtual disks into redundancy groups. For example, providing single archive fault tolerance requires that no more than one virtual disk from a single archive be placed into a RAID 5 redundancy group, while no more than two virtual disks from a single archive may be placed in to a RAID 6 group. By constructing the redundancy groups in this fashion, the number of decisions made by the placement layer is significantly decreased, since placing a set of shards from a single fragment among distinct archives is sufficient to maintain security and reliability.

As virtual disks and redundancy groups are created, parity information is propagated to the affected archives. Each archive will receive a set of shard group IDs with their newly assigned type (data or parity). The relationship between the virtual and physical entities is shown in Figure 5.

3.3.2 Information Flow

During a single client ingestion, an archive placement decision is made at the placement layer, where each shard is stored at a distinct archive. The placement layer has already divided the shards up into their respective security and reliability domains; now the archive must place the shards into redundancy groups by randomly choosing a shard group for placement. The shard placement process consists of three distinct steps. First, a random shard group is chosen as the storage location of the shard. Next, the shard is placed in the last available slot in the the shard group. Finally, the corresponding parity groups for the chosen shard group are retrieved from the recovery manager and the parity updates are sent to the proper archives. Each parity update contains the data stored in the slot and the appropriate parity slot location. The failure of any parity update will result in a roll-back of the parity updates and placement of the shard into another redundancy group. Although it is assumed that all of the archives are trusted, we are currently analyzing the security effects of passing shard data between the archives during parity updates and exploring techniques for preventing archives from accumulating shards to perform brute-force data reconstruction.

3.3.3 Reliability

Long-term reliability of user data is provided by the redundancy groups defined across multiple archives. In the event of a whole or partial archive failure, the archives collaborate to reconstruct and re-distribute the contents of the failed archive. The second, redundant split in the encoding process gives a user short-term reliability, which may be needed during temporary outages or archive reconstruction. In fact, in the general case only *m* of *n* shares are used when recovering fragments, so it is unlikely that a client will notice any differences during temporary outages.

Archive reconstruction begins with a request to a single recovery manager. The recovery manager constructs a set of messages that contain the instructions for shard group reconstruction and where to store the result. Each message contains the information required to reconstruct a set of shard groups from the failed archive. A single message traverses each archive in the failed archive's redundancy group until the contents of the shard groups are reconstructed. The reconstructed shard groups are then stored at the location chosen by the recovery manager. Each message contains an entry for each participating archive and a buffer that will eventually contain the reconstructed data. An entry consists of an archive address, an operation and a set of shard group identifiers. Each archive extracts its entry and performs one of three operations: *read*, *XOR* or *write*. If the operation is a read or an XOR, the shard group identifiers in the entry are used to either read the shard groups into the data buffer or perform the XOR sum of the shard groups and data buffer. If the operation is a write, then the current archive stores the contents of the data buffer locally, since it is the fail-over archive.

There are two distinct types of fail-over used to perform archive recovery. First, an entire archive can be replaced by another, newly added archive. In this scenario, the recovery manager writes all of the reconstructed data to the new archive and the appropriate information is updated in the redundancy group indices. If a failure occurs and a new archive has not been added to carry the load, then the recovery manager must decide how to distribute the reconstructed data. The target archive used to store reconstructed data is chosen at virtual disk granularity. The reconstructed data must be distributed among the archives in a way that does not break the fault tolerance provided by the redundancy groups. In addition, migrating the data from the failed archive to another archive in the system may result in a change of security domain. When performing this type of reconstruction, the manager instructs the reconstructed virtual disks to migrate as soon as the failed archive re-joins the system or a new archive joins the system, thus re-placing the recovered data in a safe security domain.

3.3.4 Integrity

Preserving data integrity is a critical threat to all long-term archives. As the age of a system increases, so does the chance of data degradation, so POTSHARDS provides two different forms of integrity checking. The first technique requires each of the archives to periodically check its data for integrity violations using a hash stored in the header of each shard group. The second technique is a form of inter-archive integrity checking that utilizes algebraic signatures [27] across the redundancy groups in the system to perform distributed integrity checks.

Each shard group includes a hash in its header for integrity checking purposes. It is the responsibility of each individual archive to periodically check the integrity of its data by comparing the hash to the contents of the shard group. Given the immense amount of data that is stored on a single archive, integrity checking can quickly become a daunting task [26]. An archive can either perform integrity scans periodically at some predefined interval or opportunistically before or after a shard group write. The latter case requires a background process to scan the shard groups, while the former case adds the computation of a hash and hash comparison to every shard group update.

Redundancy groups can be used for purposes other than rebuilding failed archives. Using algebraic signatures over the virtual disks of a redundancy group, the integrity of the data can be checked from any archive or a third party without giving away too much information about the data. Algebraic signatures, which have the property that the signatures of the parity equals the parity of the signatures, can also be used to verify that an archive is storing data properly [27]. This scheme can be used both for "standard" RAID and for error correcting codes that can tolerate multiple erasures, such as XOR-based Reed-Solomon. Algebraic signature requests can periodically be made by a recovery manager, where a set of archives are queried for the algebraic signature of a specific interval of data. Each archive computes its required signature and sends a response to the requesting recovery manager, which verifies the correctness of the response. This distributed check ensures that the archives are not simply throwing away data and are performing internal integrity checks. If an integrity violation occurs, then the recovery manager must determine where the violation(s) originated. Determining the appropriate position of the violation(s) is simple if the number of positions is less than or equal to the error correcting capability of the redundancy algorithm. A combinatoric search using more algebraic signature queries is required if the number of violations is beyond the error correcting capability of the redundancy algorithm.

4 Experiments

The experiments on the POTSHARDS prototype were designed to show several things: the performance of the system broken down layer-by-layer, overall system throughput as more clients write to the system, the performance of POTSHARDS in a actual global environment, an analysis of shard reconstruction, and the verification of whole archive recovery. Our experiments were conducted on POTSHARDS running on both a local and distributed environment using a variety of workloads appropriate for long-term archival storage. The workloads contained a mixture of plaintext, PDF, PS and images. The read/write performance numbers reflect the performance of POTSHARDS during normal operation, while reconstructing data from shards and recovery from a failed archive represent special cases in system state.

In our experiments, the user's computer contains the client application along with the transformation and placement layers shown in Figure 1. In the local experiments, these layers were run on systems with two Pentium 4 processors running at 2.74 Ghz with 2 GB of RAM. The operating system on each was Linux version 2.6.9-22.01.1. For local tests, there were sixteen archives, each hosted on systems with two Pentium 4 processors running at 2.74 GHz. Each system had 3 GB of RAM and 7.3 GB of available local hard drive space. The archives were all running Linux version 2.6.9-34. In all of the local tests, the hosts were located on the same local area network. To simplify the experiments, the recovery manager ran from a single host and propagated parity information to the individual archives as new updates arrived. The PlanetLab [19] experiments were run in a slice that contained 12 PlanetLab nodes distributed across the globe. Of the 12 nodes, 8 were used as archives and the remaining 4 were used to run the client application, transformation layer and placement layer.

The POTSHARDS prototype system itself consists of roughly 15,000 lines of Java 5.0 code. Communications between layers utilized Java sockets over standard TCP/IP. The archives used Sleepycat Software's Berkeley DB version 3.0 for persistent storage of shards.

4.1 Read and Write Performance

The architecture of POTSHARDS is based on a four primary components: a client, transformation layer, placement layer and archive layer. Each layer, seen in Figure 1 communicates with its adjacent layer through a series of messages. In the current implementation, the client submits blocks synchronously, awaiting a response from the system before submitting the next block. In contrast, the remainder of the system is highly asynchronous. Table 2 profiles the ingestion and extraction of one block of data. It compares the time taken on an unloaded local cluster of machines and the heavily loaded, global scale PlanetLab. In addition to the time, the table details the number of messages exchanged during the request.

As the numbers clearly show, the majority of the time on the local cluster is spent in the transformation layer. This is to be expected because polynomial generation and linear interpolation in the Shamir secret-splitting algorithm is compute-intensive. Additionally, the local cluster is interconnected by a dedicated high-throughput, low-latency network with almost no outside cross-traffic. The transformation time for ingestion is greater than for extraction for two reasons. First, during ingestion, the transformation must generate many random values. In future implementations, this could be optimized through the use of pre-generated values. Second, during extraction, the transformation layer performs linear interpolation using only those shards that are necessary. That is, given an m/n secret split, only m of the shares are used even if all n are available. During extraction, the speed improvements in the transformation layer are balanced by the time required to collect the requested shards from the archive layer.

In a congested, heavily loaded system, the time to move data through the system begins to dominate the transformation time as the PlanetLab performance figures in Table 2 show. This is evident in the comparable times spent in the transformation layers in the two environments contrasted with the very divergent times spent on requests and responses in the two environments. For example, the extraction request trip took only 28 ms on the local cluster but required about 6.5 seconds on the PlanetLab trials. Since request messages are quite small, the difference is even more dramatic in the shard acquisition times for extraction. Here, moving the shards from the archives to the transformation layer took only 832 ms on the local cluster but over 29.5 seconds on PlanetLab.

Ingestion Profile		Cluster	PlanetLab
Transformation Layer Request	time (ms)	1509	2276
	msgs in	1	1
	msgs out	1	1
Placement Layer Request	time (ms)	37	30606
	msgs in	1	1
	msgs out	6	6
Archive Layer Request	time (ms)	67	39109
	msgs in	6	6
	msgs out	6	6
Response Trip	time (ms)	88	54271
Total Round Trip	time (ms)	1731	95952

Extraction Profile		Cluster	PlanetLab
Shard Acquisition	time (ms)	832	29666
	msgs	34	34
Transformation Layer Response	time (ms)	1009	1698
	msgs in	1	1
	msgs out	1	1
Request Trip	time (ms)	28	6493
Total Round Trip	time (ms)	1843	31410

Table 2: *Profile of the ingestion and extraction of one block of data comparing trials run on a lightly-loaded local cluster with the global-scale PlanetLab. Results are the average of 3 runs of 36 blocks per run. Parameters: 2 XOR secrecy split, 2 of 3 Shamir redundancy split.*

The measurements per block represent two distinct scenarios. The cluster numbers are from a lightly-loaded, well-equipped and homogeneous network with unsaturated communication channels. In contrast, the PlanetLab numbers feature far more congestion and resource demands as POTSHARDS contended with other processes for both host and network facilities. However, in archival storage, latency is not as important as throughput. Thus, while these times are not adequate for low-latency applications, they are acceptable for archival storage.

Because the per-block time is roughly equivalent, the throughput for ingestion and extraction on one client is also roughly equal. In testing, the synchronous design of the client resulted in a per client throughput of 0.50 MB/s extraction and 0.43 MB/s ingestion. However, the high level of parallel operation in the lower layer is demonstrated in the throughput as the number of clients increases. As Figure 6 illustrates, the read and write throughput scales in a nearly linear fashion as the number

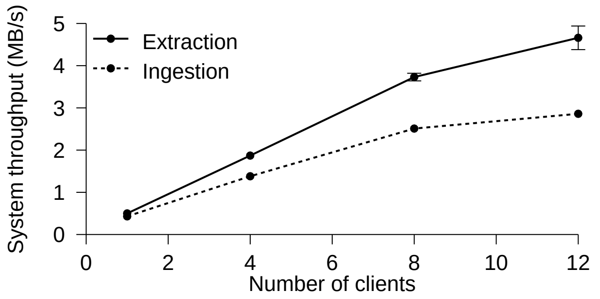
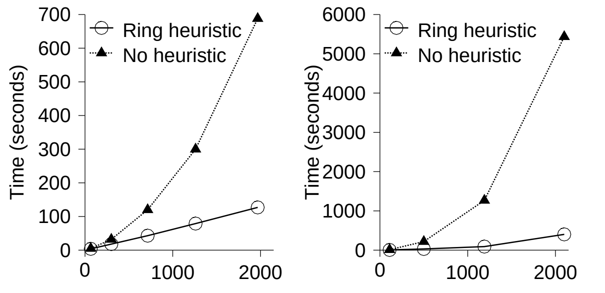


Figure 6: *System throughput for a workload of 100 MB per client. Error bars were omitted when the standard deviation of multiple runs was less than 0.05 MB/s.*



(a) 2 of 3 split. (b) 3 of 5 split.

Figure 7: *Brute force recovery time for shards generated using different secret splitting parameters.*

of clients increases. With a low number of clients, much of the system's time is spent waiting for a request from the transformation layer. Write performance is improved through the use of asynchronous parity updates. While an ingestion response waits for the archive to write the data before being sent, it does not need to wait for the parity updates. As the number of clients increases, the system is able to take advantage of the increased aggregate requests of the clients to achieve system throughput of 4.66 MB/s for extraction and 2.86 MB/s for ingestion. One goal for future work is to improve system throughput by implementing asynchronous communication in the client.

4.2 User Data Recovery

In the event that the index over a user's shards is lost or damaged, user data (including the index, if it was stored in POTSHARDS) can be recovered from the shards themselves. To begin the procedure, the user would authenticate herself to each of individual archives and obtain all of her shards. The user would then apply the algorithm described in Section 2.2 to rebuild the fragments.

We ran experiments to measure the speed of the recovery process for both algorithm options. While the recovery process is not fast enough to use as the sole extraction method, it is fast enough for use as a recovery tool. Figure 7 shows the recovery times for two different secret splitting parameters. Using the ring heuristic provides a near-linear recovery time as the number of shards increases, and is much faster than the naïve approach. In contrast, recovery without using the ring heuristic results in an exponential growth. This is very apparent in Figure 7(b), which must potentially try each path three times. The ring heuristic provides an additional layer of security because a user that can properly authenticate to all of the archives and acquire all of their shards can recover their data very quickly. In contrast, an intruder that cannot acquire all of the needed shards must search in exponential time.

Name Space	Shards	False Rings	Time
16 bits	4190	24451	6715 sec
32 bits	4190	0	225 sec

Table 3: *Recovery time in a name space with 5447 allocated names for two different name space sizes.*

The density of the name space has a large effect on the time required to recover the shards. As shown in Table 3, a sparse name space results in fewer false shard rings (none in this experiment) and is almost 30 times faster than a densely packed name space. An area of future research is to design name allocation policies that balance the recovery times with the security of the shards. One simple option would be to utilize a sliding window into the name space from which names are drawn. As the current window becomes saturated it moves within the name space. This would ensure adequate density for both new names and existing names.

4.3 Archive Reconstruction

The archive recovery mechanisms were validated on our local system using eight 1.5 GB archives. Each redundancy group in the experiment contained 8 virtual disks encoded using RAID 5. A 25 MB client workload was ingested into the system using 2 of 2 XOR and 2/3 Shamir, which resulted in 150 MB of total client data not including the appropriate parity. After the workload was ingested, an archive was failed. The recovery manager 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 archive and the reconstructed archive were compared. This procedure was run 3 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.

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	$O(R^{m-1})$	device refresh

Table 4: *Capability overview of the storage systems described in Section 5. "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.*

5 Related Work

The design concepts and motivation for POTSHARDS derive from various research projects, ranging from general-purpose distributed storage systems to distributed content delivery systems, to archival systems designed for short-term storage and archival systems designed for very specific uses such as public content delivery. However, none of these systems, a representative sample of which is summarized in Table 4, has the combination of long-term data security and proof against obsolescence that POTSHARDS provides.

Many systems such as OceanStore [16, 23], FARSITE [1], PAST [24], SNAD [18], and Plutus [14] rely on the explicit use of keyed encryption to provide file secrecy. While this may work reasonably well for short-term file secrecy, it is less than ideal for the very long-term storage problem that POTSHARDS is addressing. Further evidence that POTSHARDS is designed for a different target can be found in the design tradeoffs made in the systems mentioned previously. For example, FARSITE uses pure replication rather than erasure coding in order to provide for better read performance. In contrast, the design emphasis on POTSHARDS is reliability for very long-term storage.

Other storage projects that use distributed storage techniques but rely on keyed encryption for file secrecy do not provide any method for ensuring long-term file persistence. These systems, such as Glacier [13] and Freenet [5], are designed to deal with the specific needs of content delivery as opposed to to the requirements of long-term storage. An archival storage system must explicitly address the problem of ensuring the persistence of the system's contents.

Storage systems such as Venti [21] and Elephant [25] are concerned with archival storage but tend to focus on the near-term time scale. Both systems are based on the philosophy that inexpensive storage makes it feasible to store many versions of data. These systems, and others that employ "checkpoint" style backups, do not directly address the security concerns of the data content nor do they address the needs of long-term archival storage. Venti and commercial systems such as the EMC Centera [12] use content-based storage techniques to achieve their goals, naming blocks based on a secure hash of their data. This approach increases reliability by providing an easy way to verify the content of a block against its name.

LOCKSS [17], Intermemory [9] and other similar systems are aimed at long-term storage of open content, preserving digital data for libraries and archives where file consistency and accessibility are paramount. These systems are developed around the core idea of very long-term access for public information; thus file secrecy is explicitly not part of the design.

The PASIS architecture [11, 33], GridSharing [31], and CleverSafe [6] avoid the use of keyed encryption by using secret-splitting (k,m) threshold schemes [4, 20, 22, 28]. While this approach prevents the introduction of the singular point of failure that keyed encryption introduces to a system, these systems only use one level of secret splitting, in effect combining the secrecy and redundancy aspects of the systems. While related, these two elements of security are, in many respects, orthogonal to one another. Combining the secrecy and redundancy aspects of the system also has the possible effect of introducing compromises into the system by restricting the choices of secret splitting schemes. An earlier paper on POTSHARDS [30] discussed these mechanisms, but lacked implementation details and left many issues unanswered. By splitting secrecy and redundancy into separate mechanisms, POTSHARDS is able to implement a security mechanism optimized for redundancy or secrecy.

None of PASIS, CleverSafe, or GridSharing are designed to prevent attacks by insiders at one or more sites who can determine which pieces they need from other sites and steal those specific blocks of data. PASIS addressed the issue of refactoring secret shares [32]; however, this approach could compromise the system unless very carefully done because the refactoring process would have to read enough information in aggregate to rebuild the data. By keeping this on separate nodes, the PASIS designers hoped to avoid information leakage.

The technique for distributed rebuilding to recover from a lost archive implemented in POTSHARDS is not new, though the approach to keep distributed data secret is novel. Disaster recovery has long been a concern for storage systems [15]; Stonebraker and Schloss first introduced distributed RAID [29] to provide redundancy against site failure via geographic distribution and RAID-style algorithms. Later systems such as Myriad [3] and OceanStore [16, 23] expanded this approach to use more general redundancy techniques including m/n error correcting codes.

6 Future Work

While we have designed and implemented an infrastructure that supports secure long-term archival storage without the use of encryption, there are still some outstanding issues. POTSHARDS assumes that individual archives are relatively reliable; however, automated maintenance of large-scale archival storage remains challenging [2]. This issue is particularly critical for systems that must survive for decades to centuries; changes in basic hardware will almost certainly occur, yet the individual archives must evolve with these changes. For example, power-managed disk arrays have recently become attractive alternatives to tape [7]. We plan to explore the construction of archives from autonomous power-managed devices that can distribute and replicate storage amongst themselves, reducing the level of human intervention to replacing disks when sufficiently many have failed.

Another area of research we plan to pursue is in improving the security of POTSHARDS. Currently, POTSHARDS depends on strong authentication and intrusion detection to keep data safe, but it is not clear how to detect intrusions that may occur over many years. We are exploring approaches that can refactor the data [32] so that partial progress in an intrusion can be erased by making new shards “incompatible” with old shards. Unlike the failure of an encryption algorithm, which would necessitate wholesale re-encryption of a set of large archives as quickly as possible, refactoring for security could be done over time to limit the window over which a slow attack could succeed. We are considering integrating refactoring into the process of migrating data to new storage devices. We would also like to reduce the storage overhead in POTSHARDS, and are considering several approaches to do so. Some information dispersal algorithms may have lower overheads than Shamir secret splitting; we plan to explore their use, assuming that they maintain the information-theoretic security provided by our current algorithm.

The research in POTSHARDS is only concerned with preserving the bits that make up files; understanding the bits is an orthogonal problem that must also be solved. Lorie and others have begun to address this problem [8], 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.

7 Conclusions

This paper introduced POTSHARDS, a system designed to provide secure long-term archival storage. The long data lifetimes required of modern archival storage systems present new challenges and new security threats that POTSHARDS addresses.

In developing POTSHARDS, we made several key contributions to secure long-term data archival. First, we use multiple layers of secret splitting, approximate pointers, and archives located in independent authorization domains to ensure secrecy, shifting 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 reconstruct 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, we 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, we made intrusion detection 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, we ensure long-term data integrity 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.

Our experiments show that the current prototype implementation can store nearly 3 MB/s of user data and retrieve user data at 5 MB/s. Since POTSHARDS is an archival storage system, throughput is more of a concern than latency, and these throughputs exceed typical long-term data creation rates for most environments. The storage process is parallelizable, so additional clients increase throughput until the archives' maximum throughput is reached, and additional archives linearly increase maximum system throughput.

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 archival storage. 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.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this paper, helping us to refine them. We particularly thank Erez Zadok and Andrew Leung for reading early drafts of this paper. We also thank the sponsors of the SSRC, including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Hewlett-Packard Laboratories, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Symantec, and Yahoo.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. 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)*, Boston, MA, Dec. 2002. USENIX.

[2] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, pages 221–234, Apr. 2006.

[3] F. Chang, M. Ji, S.-T. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Jan. 2002.

[4] S. J. Choi, H. Y. Youn, and B. K. Lee. An efficient dispersal and encryption scheme for secure distributed information storage. *Lecture Notes in Computer Science*, 2660:958–967, Jan. 2003.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[6] CleverSafe. Highly secure, highly reliable, open source storage solution. Available from http://www.cleversafe.org/, June 2006.

[7] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*, Nov. 2002.

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

[9] A. V. Goldberg and P. N. Yianilos. Towards an archival intermemory. In *Advances in Digital Libraries ADL'98*, pages 1–9, April 1998.

[10] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 1999.

[11] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networking (DSN 2004)*, June 2004.

[12] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 60–71, June 2005.

[13] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005. USENIX.

[14] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, San Francisco, CA, Mar. 2003. USENIX.

[15] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Apr. 2004.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[17] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital

preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, 2005.

[18] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

[19] L. Peterson, S. Muir, T. Roscoe, and A. Klingaman. PlanetLab Architecture: An Overview. Technical Report PDN–06–031, PlanetLab Consortium, May 2006.

[20] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE)*, 27(9):995–1012, Sept. 1997. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

[21] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[22] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36:335–348, 1989.

[23] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Mar. 2003.

[24] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, Oct. 2001. ACM.

[25] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, Dec. 1999.

[26] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418. IEEE, Oct. 2004.

[27] T. Schwarz, S. J. and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)*, Lisboa, Portugal, July 2006. IEEE.

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

[29] M. Stonebraker and G. A. Schloss. Distributed RAID—a new multiple copy algorithm. In *Proceedings of the 6th International Conference on Data Engineering (ICDE '90)*, pages 430–437, Feb. 1990.

[30] M. Storer, K. Greenan, E. L. Miller, and C. Maltzahn. POTSHARDS: Storing data for the long-term without encryption. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*, Dec. 2005.

[31] A. Subbiah and D. M. Blough. An approach for fault tolerant and secure data storage in collaborative work environements. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, pages 84–93, Fairfax, VA, Nov. 2005.

[32] T. M. Wong, C. Wang, and J. M. Wing. Verifiable secret redistribution for threshold sharing schemes. Technical Report CMU-CS-02-114-R, Carnegie Mellon University, Oct. 2002.

[33] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable storage systems. *IEEE Computer*, pages 61–68, Aug. 2000.

[34] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.