POTSHARDS : Storing Data for the Long-term Without Encryption

Kevin Greenan Mark Storer Ethan L. Miller Carlos Maltzahn

Storage Systems Research Center
Computer Science Department
University of California, Santa Cruz

Abstract

Many archival storage systems rely on keyed encryption to ensure privacy. A data object in such a system is exposed once the key used to encrypt the data is compromised. When storing data for as long as a few decades or centuries, the use of keyed encryption becomes a real concern. The exposure of a key is bounded by computation effort and management of encryption keys becomes as much of a problem as the management of the data the key is protecting. POTSHARDS is a secure, distributed, very long-term archival storage system that eliminates the use of keyed encryption through the use of unconditionally secure secret sharing. A (m,n) unconditionally secure secret sharing scheme splits an object up into n shares, which provably gives no information about the object, unless m of the shares collaborate.

POTSHARDS separates security and redundancy by utilizing two levels of secret sharing. This allows for secure reconstruction upon failure and more flexible storage patterns. The data structures used in POTSHARDS are organized in such a way that an unauthorized user attempting to collect shares will not go unnoticed since it is very difficult to launch a targeted attack on the system. A malicious user would have a difficult time finding the shares for a particular file in a timely or efficient manner. Since POTSHARDS provides secure storage for arbitrarily long periods of time, its data structures include built-in support for consistency checking and data migration. This enables reliable data churning and the movement of data between storage devices.

Keywords : Data Security, Distributed Storage, Secure Storage, Survivable Storage

1 Introduction

In today's computing environment, more and more data is being migrated from hard copy to digital form. This trend is catalyzed by a number of motivations revolving around economic efficiency. Digital data offers many economic advantages compared to traditional, tangible publishing methods such as books, magazines, and films. Digital data is often much easier and cheaper to transport and store in comparison to traditional mediums such as paper and ink based printing. In many cases, digital content is also cheaper to produce. Many forms of traditional content delivery such as books and magazines are already created and edited as digital data; thus the production of a hard-copy simply adds additional costs.

Despite the advantages of digital content, tangible hard-copies of data do offer at least one major advantage over digital data. Archivists have, over many years, developed a diverse set of strategies for preserving hard-copies of data. Additionally, archivists have become very adept at judging degradation of hard-copy media through empirical observation and testing. The relative newness of computer data presents many challenges as digital archivists develop the tools and techniques to preserve digital data.

The access patterns of archival storage are distinctly different from general purpose storage. Archival storage is heavily write-centric—information is written to an archive and it may be a long time, if ever, before that data is accessed from the archive. An example of this would be a collection of business documents that must be preserved for legal reasons even though they are rarely, if ever, requested. This is the exact opposite of the model of shared storage for a distributed application or content distribution in which the access patterns would be heavily skewed towards reading or editing data For example, a web page may be written once to a storage system but read many times by many different clients. Additionally, archival storage is less concerned with throughput and latency than it is with ensuring data persistence, integrity and security.

This paper introduces the POTSHARDS (Protection Over Time, Securely Harboring And Reliably Distributing Stuff) project, an archival storage system designed for a computing environment where relatively static data must be preserved for an indefinite period of time. POTSHARDS separates data redundancy and data secrecy and utilizes a geographically distributed array of network at-

tached storage devices, called *archives*. The first phase of data storage involves the use of a secret sharing algorithm to ensure data secrecy. This avoids the problem introduced by keyed cryptography where the key represents a single point of failure which could render data recovery infeasible. This also helps avoid the problem of preserving historic keys associated with an archive of encrypted files. The second phase of data storage in POTSHARDS utilizes a data redundancy algorithm to ensure data persistence. The longevity of the data within the system is ensured through the redundancy inherent to secret sharing schemes and to aggressive consistency checking.

In order to understand the methods we propose to use in POTSHARDS, an elementary understanding of secret sharing is necessary. Although we chose to not bound POTSHARDS to use any single secret sharing algorithm, two popular algorithms will be quickly explained. We assume that POTSHARDS will be equipped to use any secret sharing scheme.

A rather simple approach to sharing a b-bit secret data block is to generate $n-1$ random b-bit blocks and XOR the blocks to the secret data block as follows: $P = rand_1 \bigoplus rand_2 \bigoplus \cdots \bigoplus rand_{n-1} \bigoplus secret$. The $n-1$ *rand* blocks and the result P could be distributed among n participants and the *secret* block could be tossed away. In this case, an attacker would need all n blocks in order to reconstruct the secret. Any number of the blocks less than n will not reveal anything about the secret. This scheme works very well for security-centered storage.

Shamir's secret sharing scheme is often called an (m,n)-threshold scheme [12, 8], since $m \leq n$ of the original n shares are needed to reconstruct the secret, where m is chosen when the shares are created. Shamir's scheme is based on polynomial generation and interpolation. First, a random polynomial of degree $m-1$ is created by generating $m-1$ random coefficients $c_1, c_2, \ldots, c_{m-1}$ and placing the secret at coefficient c_0 in the polynomial. m participants can collaborate to generate the interpolation polynomial $P_{m-1}(x)$. The secret is revealed by evaluating $P_{m-1}(0)$. If fewer than m participants collaborate, then the secret will not be revealed, since at least m of the shares are required to reconstruct the secret.

As of now, the first level of splitting requires a secret sharing scheme similar to the two schemes covered. The second level of splitting can be use any form of redundancy, such as Reed-Solomon encoding or Shamir's scheme. Obviously, using the XOR-based secret sharing scheme would not be sufficient for the second level of splitting. As we will show, each object written to POTSHARDS is subject to two levels of splitting, where *fragments* are created at the first, secure split and *shards* are a product of splitting *fragments* for redundancy.

2 Related Work

The design concepts and motivation of the POTSHARDS project borrow from various research projects. These projects range from general purpose distributed storage systems, to distributed content delivery systems, to archival systems designed for very specific uses.

A number of systems such as OceanStore [5], FarSite [1], and PAST [9] 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 application can be found in the design choices made by the authors of the systems mentioned previously. For example, in OceanStore straight replication was chosen in favor of erasure coding in order to provide for better read performance. In contrast, the design emphasis on POTSHARDS is reliability for very long-term storage.

Another class of storage projects that use distributed storage techniques but rely on keyed encryption for file secrecy do not provide any method for insuring long-term file persistence. These systems, such as Glacier [4] and Freenet [3] 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 insuring the persistence of the system's contents.

Another class of systems is aimed at long-term storage but with the explicit goal of open content. Systems such as LOCKSS [7], and Intermemory [2] are designed around preserving digital data for libraries and archive where file consistency and accessibility are paramount. These systems are developed around the central idea of very long-term access for public information and thus file secrecy is explicitly not part of the design.

The PASIS architecture [13] and the work of Subbiah and Blough [11] avoids the use of keyed encryption by using secret sharing threshold schemes. While this prevents the introduction of the singular point of failure that keyed encryption introduces to a system, the design of these system only use one level of secret sharing. In effect this combines 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 sharing schemes. By separating secrecy and redundancy, an implementation of POTSHARDS is able to utilize a security mechanism optimized for redundancy or secrecy.

3 Design Goals

3.1 Assumptions

One of the motivating ideas of the POTSHARDS project is the need for secure, very long-term storage. To this end certain assumptions are made out of the understanding that very long-term storage must take into account advances in computing technology. These advances in technology are difficult to predict, but POTSHARDS is made immune to them by specifying policy as opposed to mechanism. Five key policies are outlined below along with the assumptions related to the mechanisms that enforce the policy.

The first policy is related to authentication. POTSHARDS assumes that authentication is provided by the host system. The mechanism for this policy may be as simple as a security guard that verifies the identity of a user or it may be a much more advanced authentication system using cryptographic primitives. By specifying policy instead of mechanism this detail is left to the implementation. Part of the advantage of assuming that the system includes correct authentication is that, in a very long-term storage system, the file lifetimes may be far longer than the effective lifetime of a user account. For example, an employee of a company may store an important document in POTSHARDS, but it is unlikely that the user will still be a valid employee decades later. In this scenario, file ownership runs the risk of becoming little more than a historical side-note of file origins. The contents of POTSHARDS are designed to be protected through security policies that designate security clearance. This allows much of the problem of file access to encapsulated in the authentication layer.

The second policy is related to network traffic. POTSHARDS assumes that all communication between nodes in the system is secure. While keyed encryption is a weakness for long-term file storage, encryption is very effective for securing network traffic; network traffic might be secured through the use of session keys as is done in SSL. The nature of keyed encryption makes it very useful for short-term security of replaceable data. For example, if the session keys of a secured communication are lost, it is a relatively straightforward procedure to generate new keys and restart the communication. In contrast, if the encryption keys for an encrypted file are lost it may not be possible to recover the file in a timely manner.

The third assumption is based on the nature of computing technology. POTSHARDS assumes that failures will occur in the system. While most systems take into consideration some level of disaster recovery, the very long-term nature of POTSHARDS and unpredictable growth of technology dictate that the system must be designed to accommodate the failure of any of its subsystems. Failures might include catastrophic failure of part of the system or Byzantine failures caused by a comprised component of the system.

The fourth assumption is that POTSHARDS is designed expressively for use as an archival storage system and thus places its design emphasis upon longevity and security. It is not designed for interactive use as a low-latency file server. The design of POTSHARDS largely considers performance in the interactive time-scale a moot point. A likely usage scenario could include a user requesting a file to be delivered at a later time when processing has completed. When faced with a design compromise POTSHARDS will opt in favor or longevity and security over throughput speed.

Finally, data may be exposed if all or a subset of the archives collude. In the case that all of the archives collude, it is possible to expose all of the information stored by POTSHARDS. The archive collusion property is both necessary and potentially dangerous. When an authorized subject requests a particular object, the archives holding the shards for that object must collude, which must result in a properly reconstructed object. On the other hand, it would be unfavorable to have archives unexpectedly collude. In the case of unexpected collusion, we assume an archives main interest is colluding only when an authorized subject requests an object. In other words, an archive would gain very little by fulfilling unauthorized requests.

3.2 Security and Replication

Storing data securely is one of the most important aspects of a long-term archival storage system, and keyed encryption is a common method of storing data securely. Unfortunately, given the lifetime of the data being stored in an archival storage system, keyed encryption may not be sufficient, due to the single point of failure introduced when encrypting with a key. Keyed encryption relies on the computational effort required to determine the key. Given enough time and computing power, an adversary might be able to compute the key for a given set of data. Often times advances in technology drastically reduce the time it takes to obtain the encryption key. For example while the DES standard using a 56-bit key was considered secure in 1977 it was only 22 years later that a cooperative effort managed to locate a decryption key in less than twenty-three hours. [10] Even 128–256 bit symmetric keys might, at some future time, be trivial to break using as-yet undiscovered algorithms, quantum computers, or biologically-based computers, among other possibilities.

Some may argue that new encryption algorithms may be applied as the old ones are broken. Unfortunately, every time a new algorithm is applied, all data in the system

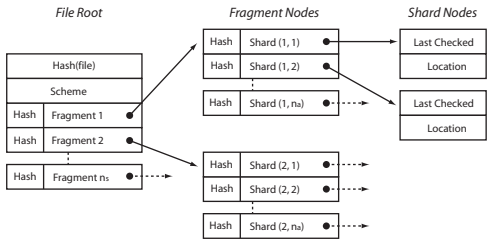

Figure 1: Shard information such as integrity data and location stored in a tree data-structure.

must be re-encrypted, which is a potential housekeeping nightmare.

In contrast, it is possible to provably store the data securely using secret sharing. Instead of relying on computational effort and the latest encryption algorithms, we can rely on the fact that an adversary would need to collect all of (or a subset of) the shares. In addition, shares can be distributed such that an adversary would have trouble effectively finding all of the shares and would not go unnoticed while launching an attack on the storage system.

Systems such as PASIS [13] combine security and redundancy using general threshold schemes. In this case, an object will be split into *n* shares, which are then given out to *n* shareholders. If any *m* shares are recovered, the original object can be reconstructed. Thus, security is accomplished by handing the shares to *n* trusted shareholders and data redundancy is accomplished by requiring only *m* of the *n* shares for reconstruction. In contrast, we aim to separate security and redundancy into two separate steps. Such a scheme will allow for the reconstruction of an object during failure without requiring knowledge of all shares created while encoding for security. Such separation also enables a system to parameterize the threshold for security and redundancy independently.

3.3 Data Structures

The shares of an object can be organized hierarchically in a tree-like structure, as shown in figure 1, which fits naturally into a two level splitting scheme. However on closer inspection this introduces several problems. For example, in a tree-like structure an object can link to its secrecy-centric fragments, which are in turn each linked to their redundancy-centric shards. In this scheme, the amount of information obtained is dependent on what level is compromised. If a fragment is compromised, then all of its shards are compromised. If the object root is compromised, then it is very possible that an adversary can reconstruct the original object.

We would like to organize the data such that its position in the data structure will not expose any information about its origin. This can be easily accomplished using a linear structure such as the one shown in figure 3. Each node in the list would have two outgoing links and two incoming links, which connect to two neighbors. Thus, if a node in the structure is compromised, then the only information exposed is that of its two neighbors. With this structure, it would be beneficial to enforce a policy that neighbor nodes not be related. An object can be reassembled by assigning a name to each shard, which allows an authorized entity to collect the correct shards and reconstruct the object.

3.4 Data Migration

We assume directed attacks will eventually occur within a long-term archival storage system. To make the task of reassembling the shards more difficult, we can make use of our data structures to churn the shards. Since the physical location of the shards does not affect object reconstruction, we can randomly migrate shards throughout the list. Such migration not only creates difficulties for directed attacks, but it can also be used as a load balancing mechanism. Migrating shards such that no single point of failure exists for an object would also be beneficial, but may be difficult to accomplish without exposing too much information about the shards. A possible solution would be to assign a system-wide failure group to each shard, which is checked upon insertion into the system's data structures. These problems will be covered in subsequent sections.

In addition to slowly churning the shards, such a system would also need to support the migration of data from one form of storage to another. As previously stated, migration between different storage devices is pivotal in POTSHARDS. We assume that failures will eventually occur and current storage technologies will one day be replaced, thus it would be to our advantage to ensure that data can be moved between any storage medium.

3.5 Malicious Attack Survivability

A key element to POTSHARDS survivability is its distributed nature. To ensure the survivability of the contents of the system, two key design elements related to the distributed nature of POTSHARDS must be enforced. These two design features relate to malicious attacks on the system.

The first design feature that must be present is that it must be very difficult to launch a targeted attack against POTSHARDS. This feature entails several aspects and is important as protection from unauthorized data access. An assumption made in this area is that activity in the system is being monitored and strange behavior can be detected and acted upon. If a malicious user is attempting to access data for which they are not authorized, the attack strategies available to the attacker should take a suffi-

ciently long time that an alarm would be raised. For example, a brute force attack where the malicious user attacks each storage node could be detected and the attacker isolated before sufficient shares are obtained to reconstruct data. Thus, the design of POTSHARDS should make it difficult to launch a time-efficient targeted attack on the system.

The second design feature that must be present is that the distributed nature of POTSHARDS must not introduce a single point of failure into the system. Any such single element would introduce a vulnerability to denial of service attacks into the system. Each of POTSHARDS subsystems and the system as a whole must be robust in design and implementation to resist an attack on any single point which could prevent access to the contents of the system.

4 Preliminary Design

The POTSHARDS design is still in the early stages of development. Even though we have not thoroughly covered all design aspects of the system, we have some idea of how the data will be organized. This section will cover techniques for splitting the data into storage units called shards, writing objects to the system, retrieving objects from the system and some of the basic data structures used for data management.

4.1 Securely Splitting the Data

The POTSHARDS system stores files as a series of fixed sized data blocks. These blocks of data are produced through two levels of data processing: the first tuned for security and the second for redundancy. The product of the first level of splitting is a set of *fragments*. These fragments are hashed to form a unique fragment identifier. Each fragment is then split into a set of *shards*, with each shard holding its source fragment's identifier. Thus, a set of shards can be used to reconstruct a fragment when a failure occurs. This process is illustrated in Figure 2.

Using two levels of splitting provides a number of interesting and useful properties. First, each object to be stored by the system can be tuned for a particular storage strategy. For example, a file that can be reproduced with relative ease but contains content that must be secure can be tuned for maximum secrecy while saving space by sacrificing a bit of redundancy. Second, in the event of a failure, an individual fragment for a file can be reconstructed without exposing any additional information about the original file. This can be very useful for online consistency checkers. If a number of shards are found to be corrupt, the remaining shards can be used to regenerate the fragment. This fragment does not expose any

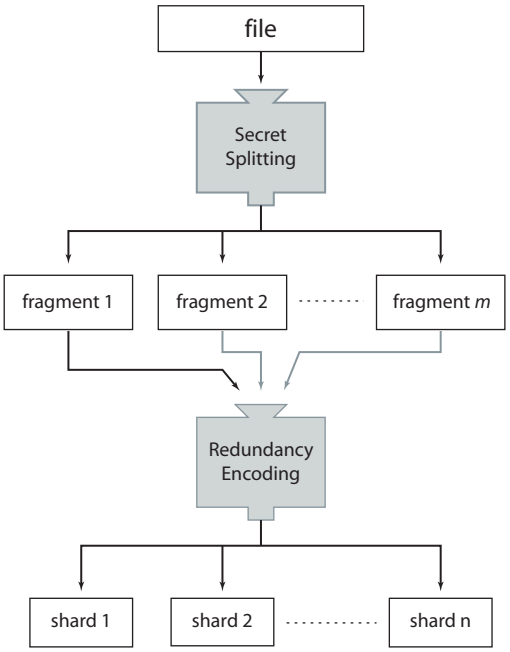

Figure 2: Splitting an object into fragments using secret splitting and fragments into shards using redundancy encoding.

information about the contents of the original file and can be used to regenerate and redistribute the shards. Since the contents of the file are not exposed in this process it could potentially be done automatically without the need for user intervention.

4.2 Fragment Identifier Lists

A list of fragment identifiers is created when an object is added to the system. This fragment identifier list is constructed during the first level of splitting. A fragment identifier is added to an object's fragment identifier list when a fragment is created for the object. In addition, as shown in Figure 2, a fragment identifier is concatenated with a shard when the shard is created during the second level of splitting. Such placement of the fragment identifiers allows one to identify the shards needed to reconstruct the fragments for a given object. The use of the concatenated fragment identifier is explained in the next section.

4.3 Storing the Shards

We propose to organize data in a distributed, circular, doubly-linked list. An example of the basic structure is given in Figure 3. As shown in Figure 4, each node in the list contains a pointer to its predecessor, a pointer to its successor, a unique identifier, a shard, and a list of fragment identifiers representing fragments constructed using

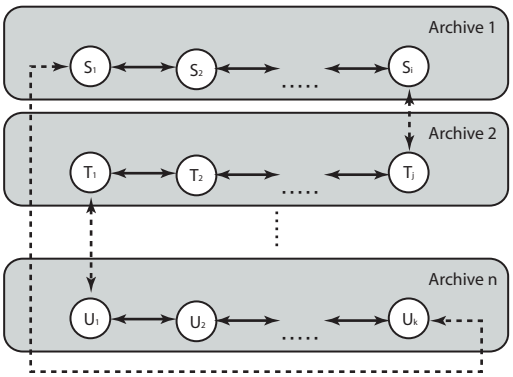

Figure 3: Data structure for organizing shares on a set of archives.

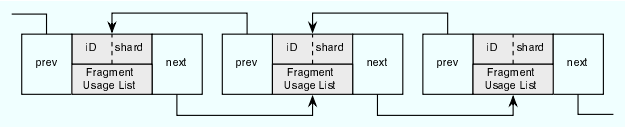

Figure 4: Individual nodes of the list containing the shards.

the contained shard. Each node in the list contains a list of fragment identifiers, because we would like to reuse shards to conserve space. Thus, it is possible for a shard to be shared between two distinct fragments. Remember to note that the shards are essentially random, thus sharing shards between fragments does not reveal any information about the object. The unique identifier is currently a cryptographic hash of the shard. Note that the cryptographic hash is used for verification at the shard level.

As shown in Figure 3, each archive holds a fraction of the list locally, where the last node on the local list points to another list on a different archive. Each node in the list contains a shard generated through the two-level splitting process, a unique identifier and a list of fragment identifiers. The contents of the list are probabilistically churned periodically to ensure that the nodes are distributed as uniformly as possible throughout the list. The contents of the nodes can be churned by specialized processes and during normal operations (i.e. searches, inserts, etc.). It is assumed that all operations performed on an archive are properly authenticated. Thus, in order to traverse the entire list, the subject traversing the list will have to authenticate with every archive in the system multiple times, once for each shard. Since the shards are periodically churned, it would be very difficult for an adversary to efficiently reconstruct any of the objects. The details involving the functionality of each archive is left to future work.

Given the structure of the data within the system, there are currently two methods of verification. First, since a node's identifier is a hash of the shard, it can be used to verify the contents of each node in the list. A more powerful method of verification involves the concatenation of the fragment identifier with the fragment before performing the second level of splitting. If the shards are created with the fragment identifier embedded, verification can occur on the fragment level. A process can collect the shards for a randomly chosen fragment by searching the list for a particular fragment identifier. The fragment identifier constructed after combining the shards would then simply be compared to the search key used when collecting the shards. Note that by randomly reconstructing fragments for verification purposes, no information about any of the objects is revealed, since a fragment only represents a single piece of many needed to actually reconstruct the object.

4.4 Creating Objects in the System

As stated in the previous section, two levels of splitting will break an object up into many pieces, called shards. We must now concern ourselves with how to store these pieces in an efficient manner. Simply storing the shards created in the second level of splitting can provide an efficient and straightforward storage solution. With reference to Figure 2, the fragments are thrown away and the shards and their respective fragment identifiers are handed off to the storage layer of the system. The shards are then inserted into the distributed, doubly-linked list in a probabilistic manner.

Access to the original contents of an object will require all of the fragments generated in the first level of splitting. Since only the fragment identifiers and shards are stored, the fragments must be reconstructed from the shards. Thus, authorization to first obtain the fragment identifier list of an object is necessary along with the authorization for the system to reconstruct a fragment using each fragment identifier. Without authorization, an adversary can only guess which fragments are used to create an object. Such a search of the system would not only require a great deal of time and computing power, but it would also be easy to detect.

4.5 Retrieving Objects from the System

As implied from the structure illustrated in the previous sections, the only piece of information necessary for immediate object reconstruction is the fragment identifier list. As of now, we are unsure how the actual fragment identifier list will be stored. We assume a subject will request an object, which will require some form of authentication. If the subject is authorized to access the object, the system will retrieve the fragment identifier list and issue a fragment reconstruction request for each fragment identifier. Since each shard is stored with a list of the fragment identifiers that use the shard, a traversal of the entire distributed list is required for each fragment identifier.

Depending on the chosen storage policies for an object, a fragment is reconstructed after all ((n,n)-scheme) or a subset ((m,n)-scheme) of the shards for the given fragment are found. The fragments are then used to reconstruct the object.

5 Open Problems and Future Work

The POTSHARDS project is still at a relatively early stage and thus many of its design elements are still in their formative stages. In fact, none of the aspects of the POTSHARDS are at a stage where their design can be considered finalized for even the initial implementation. There are still many questions that, while identified, have yet to be examined in greater detail. Some of the more pressing issues that we identified are listed below.

5.1 Data Structures

We expect POTSHARDS data structures to change as our design of the system as a whole matures. Currently, we are designing the system with doubly-linked lists in mind. An early sketch of the system used tree structures which may have improved performance but presented too much of a security risk.

Aside from the fundamental data structures, the contents of each node are subject to change as well. Some possibilities for changes to the structure of the nodes include fields to indicate status of the node. This might include fields used by garbage collection or fields which identify the nodes as being of a particular data type. The latter example could be useful if multiple types of data are stored in the list besides shards such as naming information or object metadata.

5.2 Consistency Checking

A critical aspect of very long-term storage is insuring that file consistency is maintained. Malicious users, degradation and faulty writes can all cause trouble for a system aimed at maintaining data for an extended period of time; such a system must provide a proactive solution to insuring that the integrity of its contents is protected. One method of ensuring this in POTSHARDS is through the use of active consistency checking.

One straightforward method of active consistency checking would be to check the integrity of the system contents at regular times. This would require securely recording consistency information, such as a hash value, for each shard. This method could be optimized by throttling these integrity operations to reduce overhead during times of high activity. Further optimization might involve smart checking that does not check each shard but rather checks enough shards so that the fragment could be regenerated.

A promising area of consistency checking could be the use of algebraic signatures such as those described by Litwin and Schwarz [6]. These structures could be used to optimize the consistency checking within the system compared to traditional hashing algorithms.

5.3 Archive Recovery

The distributed nature of POTSHARDS introduces the possibility of a failed storage device. Since the system is designed to provide storage for decades or longer, the failure of one storage devices or even an entire archive is inevitable. Further pressing the need for reliable disaster recovery is the doubly linked list structure used in POTSHARDS as shown in Figure 3. Since all the storage devices are connected, the loss of one device must not render the list irreparable. POTSHARDS must have a reliable way to recover from the simultaneous failures of multiple storage archives.

In addition to straightforward storage device failures, the POTSHARDS system should be able to deal with Byzantine failures where a device may be acting maliciously. While the projected implementation of POTSHARDS would consist of a controlled network of distributed devices, as opposed to a federated storage system such as FARSITE [1], the possibility still exists that a compromised storage device could be acting maliciously.

5.4 Naming

At the present time the naming of shards within the system has not been finalized. There are a number of possibilities that are being examined ranging from randomly generated names, names based on cryptographic hashes of the shards contents, or magic numbers generated by some deterministic process. Hash based naming has the advantage that the naming and consistency information is one and the same. If a malicious user has the ability to change shard data then there are two possible attack scenarios. In the first, a malicious user changes the data but not the name. In this case regenerating the name reveals the change. In the second scenario a malicious user changes the data and the name. In this case a search for a shard by name then the search simply returns a negative result.

The issue of naming extends beyond shards to other elements in the system. The methods and role of naming in dealing with fragments and objects is another area to be examined. The distributed nature of POTSHARDS also suggests that the naming of components is an important issue as well.

5.5 Storage Protocol

In the current discussion of POTSHARDS, the actual storage devices are referenced in rather general terms. As mentioned previously, the system is designed to be hardware agnostic so that it can accommodate future advances in computing technology. None the less, the capabilities and high level storage protocols must be defined so that an implementation can make adequate hardware decisions.

5.6 Migration

As POTSHARDS is designed for very long-term storage, migration is an important consideration. The distributed nature of the system suggests that archives will be coming on-line as well as leaving the system. A method of safely moving shards off of the archive that is scheduled to be removed from the system is therefore very important. Related to graceful removal of archives from the system is a method of dealing with catastrophic failure of a archive.

For archives that come on line after the initial start of the system, a method of normalizing population across all the archives in the system will be important. This may involve placing more shards on new archives, moving existing shards to new archives or a hybrid of both approaches.

Another area of migration that is being considered is the idea of data churning. Data churning would involve the automatic movement of shards within the system. This has the possible benefit of making targeted attacks more difficult. One possible difficulty is that any churning strategy would need to ensure that all of the shards needed to reconstruct an object are not inadvertently moved to the same archive. If the churning strategy is based on an even probability distribution, the chances of this occurring should be acceptably low. Even with a low probability of shard consolidation within a single archive, we would still prefer to not take any chances. We are considering policies to bound the number of single-object shards placed on an archive. For instance, some fraction f of the total number of shards for an object will place a firm upper bound on shard consolidation within a single archive.

5.7 Managing Storage Growth

If the growth of storage is not properly addressed, it is obvious that POTSHARDS may incur a great deal of storage overhead, which is the cost for long-term data protection and integrity. While the POTSHARDS system's distributed nature allows for easy installation of additional storage, efficient use of existing storage might be achieved through the use of garbage collection and shard reuse.

A possible problem with garbage collection would be designing a strategy that does not violate any of the POTSHARDS design principles. Specifically, any garbage collection strategy would have to be secure against a malicious user. For example, if a straightforward strategy of reference counting was used, the data structures must be secure from a malicious user that attempts to artificially reduce the reference count in order to trigger a clean-up. If certain shards are used for the regeneration of more than one file, as previously suggested, this sort of attack could be very damaging.

Storage efficiency might also be accomplished by reusing preexisting shards to limit the amount of storage overhead needed. In this strategy, instead of generating all the shards for a given fragment, it might be possible to use a mixture of pre-existing and randomly generated shards. The implications of this strategy would be that some shards would be used in the regeneration of more than one file. This could have the desired effect of reducing the total amount of storage overhead imposed by the secret splitting and redundancy encoding.

6 Conclusions

Current systems do not address the needs of a system that must store files securely for a very long period of time. When storing files for spans of time measured in decades, many of the common conventions used in storage introduce unacceptable weaknesses. Keyed encryption introduces a single point of failure and is only computationally bound; history has shown that this approach often fails over time. Similarly, user accounts and file ownership may be shorter lived than the files when dealing with data that must survive longer than the users that created it.

The POTSHARDS project aims to provide file security for very long-term storage through the use of secret sharing. Objects that are to be stored in the system are split into fragments in a security layer and shards in the redundancy layer. These shards are stored within a distributed storage environment in a linked list structure. Since computing technology has the potential to drastically change over several decades the POTSHARDS system is designed with a modular structure that allows for components to be upgraded over time.

Moving forward, the primary focus for the POTSHARDS system is to continue to revise and formalize the design, with the goal of producing a working prototype. A testable prototype should further force the revision of the designs described in this paper and provide some evidence for the scalability and feasibility of the system.

Acknowledgments

The authors would like to thank Owen Hofmann, Carl Lischeske, Kristal Pollack, Deepavali Bhagwat, Lawrence

You, and Darrell Long for their help in early discussions on the POTSHARDS design. Other members of the Storage Systems Research Center were also helpful. We would also like to thank the industrial sponsors of the SSRC, including Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Veritas, and Yahoo for their generous support.

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] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[3] 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.

[4] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, 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 Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Nov 2000.

[6] W. Litwin and T. Schwarz. Algebraic signatures for scalable distributed data structures. Technical Report CERIA Technical Report, Université Paris 9 Dauphine, Sept. 2002.

[7] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, M. Baker, and Y. Muliandi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of the Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct 2003. ACM.

[8] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[9] 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*, Chateau Lake Louise, Banff, Alberta, Canada, Oct 2001.

[10] D. R. Stinson. *Cryptography Theory and Practice*. Chapman & Hall/CRC, Boca Raton, FL, second edition, 2002.

[11] 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 Survivabiilty*, pages 84–93, Fairfax, VA, Nov. 2005.

[12] 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.

[13] 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.