POTSHARDS—A Secure, Recoverable, Long-Term Archival Storage System

5

MARK W. STORER, KEVIN M. GREENAN, and ETHAN L. MILLER
University of California, Santa Cruz
and
KALADHAR VORUGANTI
NetApp

Users are storing ever-increasing amounts of information digitally, driven by many factors including government regulations and the public's desire to digitally record their personal histories. Unfortunately, many of the security mechanisms that modern systems rely upon, such as encryption, are poorly suited for storing data for indefinitely long periods of time; it is very difficult to manage keys and update cryptosystems to provide secrecy through encryption over periods of decades. Worse, an adversary who can compromise an archive need only wait for cryptanalysis techniques to catch up to the encryption algorithm used at the time of the compromise in order to obtain "secure" data. To address these concerns, we have developed POTSHARDS, an archival storage system that provides long-term security for data with very long lifetimes without using encryption. Secrecy is achieved by using unconditionally secure secret splitting and spreading the resulting shares across separately managed archives. Providing availability and data recovery in such a system can be difficult; thus, we use a new technique, approximate pointers, in conjunction with secure distributed RAID techniques to provide availability and reliability across independent archives. To validate our design, we developed a prototype POTSHARDS implementation. In addition to providing us with an experimental testbed, this prototype helped us to understand the design issues that must be addressed in order to maximize security.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*

General Terms: Design, Experimentation, Performance, Reliability, Security

Additional Key Words and Phrases: Archival storage, secret splitting, approximate pointers

This research was supported by the Petascale Data Storage Institute under Department of Energy award FC02-06ER25768. Additional support for the Storage Systems Research Center was provided by Agami Systems, Data Domain, Hitachi, and by SSRC sponsors including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Diginsense, Hewlett-Packard Laboratories, IBM Research, Intel, LSI Logic, Microsoft Research, Network Appliance, Seagate, Symantec, Yahoo.
Author's address: M. W. Storer (corresponding author), K. M. Greenan, E. L. Miller, University of California at Santa Cruz, 1156 High Street, Santa Cruz, CA 95064; email: mstorer@cs.ucsc.edu; K. Voruganti, NetApp, 495 East Java Drive, Sunnyvale, CA 94089.

DOI 10.1145/1534912.1534914 http://doi.acm.org/10.1145/1534912.1534914

ACM Reference Format:
Storer, M. W., Greenan, K. M., Miller, E. L., and Voruganti, K. 2009. POTSHARDS—A secure, recoverable, long-term archival storage system. ACM Trans. Storage 5, 2, Article 5 (June 2009), 35 pages. DOI = 10.1145/1534912.1534914 http://doi.acm.org/10.1145/ 1534912.1534914.

1. INTRODUCTION

Businesses and consumers are becoming increasingly conscious of the value of archival data. In the business arena, data preservation is often mandated by law [104th Congress 1996; Oxley 2002], and data mining has proven to be a boon in shaping business strategy. For individuals, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies, and personal documents. Further, this information often needs to be stored securely; data such as medical records and legal documents that could be important to future generations must be kept indefinitely but must not be publicly accessible.

The goal of a secure, long-term archive is, therefore, to provide security for relatively static data with an indefinite lifetime. More specifically, a secure archive seeks to provide three long term features: secrecy, recoverability, and integrity. The first, long term secrecy, aims to ensure that the data stored must only be viewable by authorized readers. The second, recoverability, is akin to availability and stipulates that data must be available and accessible to authorized users within a reasonable amount of time, even to those who might lack a specific key. The third, integrity, ensures that the data read is the same as the data written.

The usage model of archival storage is quite different from that of traditional storage systems. The workload is write-heavy, motivated by regulatory compliance and the desire to save any data that *might* be valuable at a later date. Reads, while relatively infrequent, are often part of a query or audit and thus are likely to be temporally related. Thus, archival storage is more concerned with throughput than low-latency data access. Long term archives also add the unique "digital time-capsule" property that the reader may have little knowledge of the system's contents, and no contact with the original writer; while file lifetimes may be indefinite, user lifetimes certainly are not.

While storage security has long been an active, well-researched area, the indefinite lifetimes of archival storage introduce a number of new challenges [Baker et al. 2006; Storer et al. 2006]. One of the biggest challenges is that mechanisms such as cryptography work well in the short term, but are less effective in the long term. The use of computation-bound encryption in an archival scenario introduces the problems of lost keys, compromised keys, and even compromised cryptosystems. All this is exacerbated by the numerous key rotations and cryptosystem migrations that will inevitably occur over the course of several decades; this must all be done without user intervention because the user who stored the data may be unavailable. Thus, security for archival storage must be designed explicitly for the unique demands of long term storage.

To address the many security requirements for long term archival storage, we have designed and implemented POTSHARDS (Protection Over Time, Securely

Harboring And Reliably Distributing Stuff), which uses three primary techniques to provide security for long term storage. First, secret splitting [Shamir 1979] is used to produce a tuple of *n secret shares* from a block of data, *m* of which must be obtained to reconstruct the block. Unlike encryption, secret splitting is unconditionally secure; it can be shown that combining fewer than *m* shares reveals *no* information about the original block. Second, POTSHARDS uses a global data namespace that is used to identify data entities. The namespace introduces an element of diversity [Forrest et al. 1997] into the model, since it is sparsely populated and treated in a similar fashion to a heap. Third, POTSHARDS utilizes approximate pointers, which differ from traditional pointers in that they indicate a *region* in the namespace as opposed to an exact address. Approximate pointers enable secure recovery from only the data itself by associating related secret shares.

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

The recovery and availability strategy of POTSHARDS enables the reconstruction of data from the secret shares alone. Thus, even with no outside index to connect data blocks and secret shares, a user's data can be recovered. This is especially important in long term archival scenarios in which data may have a potentially indefinite lifetime [Baker et al. 2006; Storer et al. 2006]. Our approach is based upon the use of approximate pointers, which provide clues about inter-share relationships. These clues supply enough information to allow recovery but require a lot of shares, a necessity that is difficult for an adversary to meet.

In order to validate our design and test its performance, we implemented a prototype of POTSHARDS. The current implementation can read and write data at a rate of 4–6 MB/s on commodity hardware. It also survives the failure of an entire archive with no data loss and little effect seen by users. In addition, we demonstrated the ability to rebuild a user's data from all of the user's stored shares without the use of a user index. Finally, our implementation provides a testbed we have used to perform a thorough exploration of our security model. These experiments demonstrate the system's suitability to the unique usage model of long term archival storage.

2. BACKGROUND

Since POTSHARDS was designed specifically for secure, long term storage, we identified three basic design tenets to help focus our efforts. First, we assume

that encrypted data can be read by anyone, given sufficient CPU cycles and advances in cryptanalysis. Put another way, if an attacker obtains encrypted data, the plaintext will eventually be revealed. Second, for long term survivability, data must be recoverable without any information from outside the set of archives; fulfilling requests in a reasonable time cannot require any outside data, such as external indexes or encryption keys. Third, we assume that individuals are more likely to be malicious than an aggregate. Thus, our system trusts groups of archives, even though it does not trust individual archives. The chances of every archive in the system colluding maliciously is small; the system allows rebuilding of stored data if all archives cooperate.

In designing POTSHARDS to meet these goals, we used concepts from a number of existing storage systems that satisfy some of the design tenets discussed before. These ranged 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 (a representative sample of these systems is summarized in Table I).

2.1 Archival Storage Models

Storage systems such as Venti [Quinlan and Dorward 2002] and Elephant [Santry et al. 1999] 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. Other systems, such as Glacier [Haeberlen et al. 2005], are designed to take advantage of the underutilized client storage of a local network. These systems, and others that employ "checkpoint-style" backups, address neither the security concerns of the data content nor the needs of long term archival storage. Venti, BitVault [Zhang et al. 2007], and commercial systems such as the EMC Centera [Gunawi et al. 2005] 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. As with the short term storage systems described before, security is often ensured by encrypting data using standard encryption algorithms.

Some systems, such as LOCKSS [Maniatis et al. 2005] and Intermemory [Goldberg and Yianilos 1998], 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; file secrecy is explicitly not part of the design. Rather, the systems exchange information about their own copies of each document to obtain consensus between archives, ensuring that a rogue archive does not "alter history" by changing the content of a document that it holds.

2.2 Storage Security

Many storage systems seek to enforce a policy of secrecy for their contents. Two common mechanisms for enforcing data secrecy are encryption and secret splitting.

Table I. Capability Overview of the Storage Systems Described in Section 2

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	cont. relocation
Publius	encryption	password (delete)	retrieval based	m (out of n)	
SNAD / Plutus	encryption	encryption	hashing	1	
SafeStore	encryption	certificates	hashing	m (out of n)	
GridSharing	secret splitting		replication	1	
PASIS	secret splitting		repair agents, audits	m (out of n)	
CleverSafe	info. 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

"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.

2.2.1 *Secrecy via Encryption.* Many systems such as OceanStore [Rhea et al. 2003], FARSITE [Adya et al. 2002], SNAD [Miller et al. 2002], Plutus [Kallahalla et al. 2003], e-Vault [Iyengar et al. 1998] and SafeStore [Kotla et al. 2007] address file secrecy but rely on the explicit use of keyed encryption. While this may work reasonably well for short term secrecy needs, it is less than ideal for the very long-term security problem that POTSHARDS is addressing. Encryption is only computationally secure and the struggle between cryptography and cryptanalysis can be viewed as an arms race. For example, a DES encrypted message was considered secure in 1976; just 23 years later, in 1999, the same DES message could be cracked in under a day [Stinson 2002]; future advances in quantum computing have the potential to make many modern cryptographic algorithms obsolete.

The use of long-lived encryption implies that reencryption must occur to keep pace with advances in cryptanalysis [Troncoso et al. 2008]. To prevent a single archive from obtaining the unencrypted data, reencryption must occur over the old encryption, resulting in a long key history for each file. Since these keys are all external data, a problem with any of the keys in the key history can render the data inaccessible when it is requested.

Keyed cryptography is only computationally secure, so a compromise of an archive of encrypted data is a potential problem regardless of the encryption algorithm that is used. An external adversary who compromises an encrypted archive need only wait for cryptanalysis techniques to catch up to the encryption used at the time of the compromise. The data's existence on a secure, long term archive suggests that data will still be valuable even if the malicious user must wait several years to read it. The situation is more dramatic when dealing with a privileged inside attacker that can decrypt any desired information, even if the data is subsequently reencrypted by the archive; an insider will have access to the new key by virtue of his internal access.

Some content publishing systems utilize encryption, but its use is not motivated solely by secrecy. Publius [Waldman et al. 2000] utilizes encryption for write-level access control. Freenet [Clarke et al. 2001] is designed for anonymous publication and encryption is used for plausible deniability over the contents of a user's local store. As with secrecy, the use of encryption to enforce long-lived policy is problematic due to the mechanism's computationally secure nature.

2.2.2 *Secrecy via Splitting.* To address the issues resulting from the use of encryption, several recent systems including PASIS [Wylie et al. 2000; Goodson et al. 2004] and GridSharing [Subbiah and Blough 2005] have used or suggested the use [Storer et al. 2005] of *secret splitting* schemes [Shamir 1979; Rabin 1989; Plank 1997; Choi et al. 2003]; a related approach used by Mnemosyne [Hand and Roscoe 2002] and CleverSafe [CleverSafe 2006] uses encryption followed by information dispersal (IDA) to attempt to provide security. In secret splitting, a secret is distributed by splitting it into a set number n of shares such that no group of k shares ($k < m \leq n$) reveals any information about the secret; this approach is called an m of n threshold scheme. In such a scheme, any m of the n shares can be combined to reconstitute the secret; combining fewer than m

shares reveals *no* information. A simple example of an n of n secret splitting scheme for a block B is to randomly generate $X_0, \ldots, X_{n-2}$, where $|X_i| = |B|$, and choose X_{n-1} so that $X_0 \oplus \cdots \oplus X_{n-2} \oplus X_{n-1} = B$. Secret splitting satisfies the second of our three tenets, that is, data can be rebuilt without external information, but it can have the undesirable side-effect of combining the secrecy and redundancy aspects of the systems. Although related, these two elements of security are, in many respects, orthogonal to one another. Combining these elements also risks introducing vulnerabilities into the system by restricting the choices of secret splitting schemes.

While secret splitting can provide the benefit of unconditionally security, it does incur a number of drawbacks compared to encryption. First, secret splitting schemes have a high storage overhead; in an m of n threshold scheme with perfect secrecy, no information can be obtained from the possession of less than m shares, but splitting results in a storage blowup of n. Second, while some XOR-based secret splitting schemes have been shown to provide fast enough throughput for low-latency access situations [Subbiah and Blough 2005], many others utilize computationally expensive operations such as linear interpolation. Fortunately, in many archival scenarios, long-term security is a valid trade-off for a reasonable increase in access time.

To ensure that our third design tenet is satisfied, a secure long term storage system must ensure that an attempt to breach security will be noticed by *somebody*, ensuring that the trust placed in the collection of archives can be upheld. Existing systems do not meet this goal because the secret splitting and data layout schemes they use are minimally effective against an inside attacker that knows the location of each of the secret shares. None of PASIS, CleverSafe, or GridSharing is 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, enabling a breach of secrecy with relatively minor effort. This problem is particularly difficult given the long time that data must remain secret, since such breaches could occur over years, making detection of small-scale intrusions nearly impossible. PASIS addressed the issue of refactoring secret shares [Wong et al. 2002]; however, this approach could compromise data in the system because the refactoring process may reveal information during the reconstruction process that a malicious archive could exploit. By keeping this on separate nodes, the PASIS designers hoped to avoid information leakage. Mnemosyne used a local steganographic file system to hide chunks of data, but this approach is still vulnerable to rapid information leakage if the encryption algorithm is compromised; the IDA provides no additional protection to the distributed pieces.

2.3 Application Security

In the area of application security, a lot of effort has gone into increasing the amount of work an adversary must perform through the introduction of randomness [Forrest et al. 1997; Bhatkar et al. 2003]. Attacks that rely upon memory exploits are often based on the fact that information is stored in very predictable and consistent locations. Randomization forces an adversary to analyze each

copy of the program they are attacking. The use of approximate pointers extends the idea of diversity into the area of data protection.

Another strategy utilized in application security involves detecting malicious behavior by utilizing attack signatures. In this technique, observed behavior patterns are matched to known attacks. To this end, signature generation tools [Wang et al. 2006; Xu et al. 2005; King and Chen 2003] are an important aid in automatically detecting the future occurrence of known attacks. The sparse namespace borrows the idea of attack detection as means to prevent data loss. However, an important difference between application protection and data protection is that a user's data is unique and personal. Thus, even one compromise could mean that unique data has been lost.

2.4 Disaster Recovery

With long data lifetimes, hardware failure is a given; thus, dealing with a failed archive is inevitable. In addition, a long term archival storage solution that relies upon multiple archives must be able to survive the loss of an archive for other reasons, such as business failure. Recovering from such large-scale disasters has long been a concern for storage systems [Keeton et al. 2004]. To address this issue, systems such as distributed RAID [Stonebraker and Schloss 1990], Myriad [Chang et al. 2002], and OceanStore [Rhea et al. 2003] use RAID-style algorithms or more general redundancy techniques including (m, n) error correcting codes along with geographic distribution to guard against individual site failure. Secure, long term storage adds the requirement that the secrecy of the distributed data must be ensured at all times, including during disaster recovery scenarios.

3. POTSHARDS OVERVIEW

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

Users access the system through a POTSHARDS client, which has three primary functions. First, the client handles all *data transformation* duties. For writes, as shown in Figure 1, this involves generating *shards* from *objects* through the use of secret splitting techniques. For reads, the process is reversed, shard identifiers are used to fetch shards from the archives, and objects are reconstructed. Second, the client is responsible for distributing shards to archives such that no single archive has enough shards to reconstruct data. Third, as the client resides on a system separate from the shards, the POTSHARDS client is responsible for handling communication between the user and the archives. The advantage of this arrangement is that data never reaches an archive in

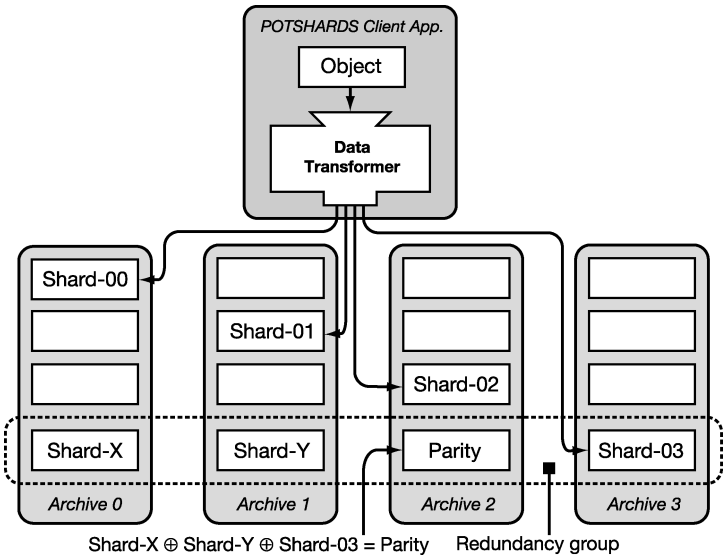


Fig. 1. An overview of POTSHARDS showing the data transformation component of the client application producing shards from objects, and distributing them to independent archives. The archives utilize distributed RAID algorithms to securely recover shards if an archive is lost.

unsecured form, and multiple CPU-bound data transformation processes can generate shards in parallel for a single set of physical archives. Of course, as in any security application, careless implementation of the POTSHARDS client can introduce unforeseen compromises; adversaries can take advantage of carelessly cached passwords and other such key material.

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

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

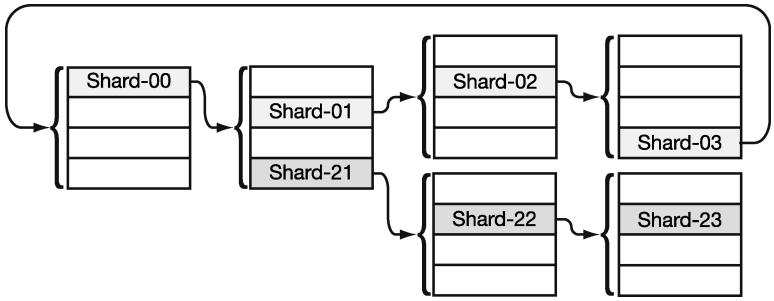


Fig. 2. Approximate pointers point to R "candidate" shards ($R = 4$ in this example) that might be next in a valid shard tuple. Shards_{0X} make up a valid shard tuple. If an intruder mistakenly picks shard_{21}, he will not discover his error until he has retrieved sufficient shards and validation fails on the reassembled data.

3.1 Security Techniques

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

Secret splitting provides the secrecy in POTSHARDS with a degree of future-proofing; it can be proven that an adversary with infinite computational power cannot gain any of the original data, even if an entire archive is compromised. Further, these algorithms provide file secrecy without the need for the key and algorithm rotations that traditional encryption introduces; perfect secret splitting is unconditionally secure. Thus, POTSHARDS is not forced into maintaining complex key histories.

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

In addition to uniquely identifying data entities in POTSHARDS and improving attack detection, the global namespace enables the use of secret splitting algorithms by imposing an ordering over entities. Many threshold schemes, such as those that rely on linear interpolation [Shamir 1979], require both the shares and a specific ordering of those shares for reconstruction. Preserving the ordering over a tuple of shards is easily accomplished by naming the shards in ascending order, according to their location within the full shard tuple. In this way, names impose a total ordering over a complete tuple of shards.

3.2 Reliability and Availability Techniques

POTSHARDS provides reliability and availability through two distinct recovery strategies. First, as Figure 2 illustrates, the shards that reconstruct a data

block form a circularly-linked list, allowing a specific user's data to be recovered from her shards alone. This ring of shards is generated within the transformation components, as part of the ingestion process. Second, the loss of an entire archive is handled using distributed RAID techniques, across multiple independent archives. This two-level approach allows POTSHARDS to scale the recovery to the size of the data loss.

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

Approximate pointers enable the use of secret splitting by providing a built-in method of "key recovery"; knowing which secret shares to combine is analogous to an encryption key because it is the secret that transforms ciphertext into plaintext. Without the clues provided by approximate pointers, recovery involves testing every possible combination of shards, making it an intractable problem. In contrast, while direct pointers would make recovery trivial, it would also compromise security; an adversary with one shard could easily make targeted attacks for the rest of the shards. Thus, the advantage of approximate pointers is that, by indicating a region and utilizing namespace sparseness, targeted attacks are much more difficult, and brute-force attacks would be quite noticeable. Thus, secrecy is not unduly affected, providing a worthwhile trade-off for slower recovery times if a block's shard list is lost.

To deal with larger-scale losses, the archive layer in POTSHARDS consists of independent archives utilizing secure, distributed RAID techniques. As Figure 1 shows, archive-level redundancy is computed across sets of *unrelated* shards, so redundancy groups provide no insight into shard reassembly. POTSHARDS includes two novel modifications beyond the distributed redundancy explored earlier [Stonebraker and Schloss 1990; Chang et al. 2002]. The first is a secure reconstruction procedure, described in Section 4.3.1, that allows a failed archive's data to be regenerated in a manner that prevents archives from obtaining additional shards during the reconstruction; shards from the failed archive are rebuilt only at the new archive that is replacing it. Second, POTSHARDS uses algebraic signatures [Schwarz and Miller 2006] to ensure intraarchive integrity as well as interarchive integrity. Algebraic signatures have the desirable property that the parity of a signature is the same as the signature of the parity.

4. IMPLEMENTATION DETAILS

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

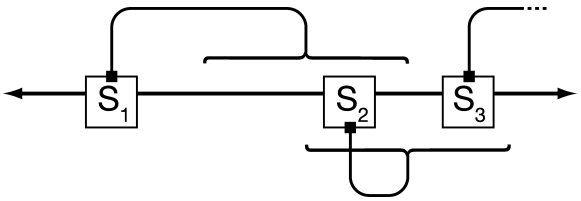


Fig. 3. Example of a situation in which careless naming has reduced the search space indicated by an approximate pointer. If shard ordering is not randomized, an adversary would know that S_3 must be greater than S_2 and thus would only need to search the region above S_2.

user's private index. Finally, we describe how approximate pointers are used to recover a user's data from the shards alone.

4.1 Naming

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

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

Careless naming and ring formation can inadvertently provide an attacker with information that effectively reduces the search space for the next shard. For example, if the shards in a tuple are ordered $S_1, S_2, \ldots, S_n$ and shard S_i always points to shard S_{i+1}, an attacker would know that the name of the next shard must be greater than the current shard. Now suppose that shard S_i itself is within the range indicated by the approximate pointer to S_{i+1}. As illustrated in Figure 3, the attacker would know that $S_i < S_{i+1}$, and thus can narrow down the search space.

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

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

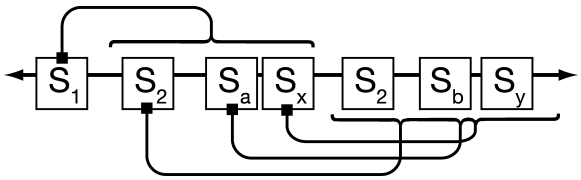


Fig. 4. Example of a situation in which careless naming has underutilized the potential of approximate pointers to increase the fan-out of linked shards. Ideally, S_2, S_a, and S_x would all point to different regions.

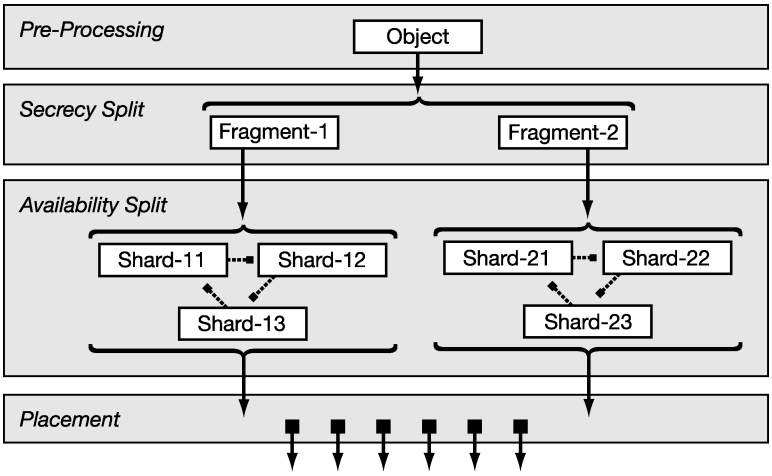
Another danger involves the underutilization of the fan-out that can be achieved with approximate pointers. Since approximate pointers indicate a region as opposed to a single address, they have the potential to greatly increase an adversary's workload. An ideal arrangement is achieved if each shard in a given region points to a different region. In this scenario, the adversary would need to acquire each shard in each of those diverse regions. Figure 4 illustrates an example in which the shard names and approximate pointers are configured poorly, resulting in little fan-out. The effect is a greatly reduced workload for the adversary; the attacker would only need to acquire the shards of overlapping regions once, rather than having to steal a given shard once for each predecessor that could point to it.

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

4.2 POTSHARDS Client: Data Transformation

One of the primary tasks of the POTSHARDS client is to perform the data transformation that produces shards from user data. As Figure 5 illustrates, the client is composed of four layers, and utilizes three unique data entities. During the ingestion of data, the preprocessing layer is responsible for producing fixed-size *objects* from user files. Objects are then transformed into *fragments* in a secret split tuned for secrecy. A second split occurs, this time tuned for availability, which transforms fragments into *shards*. Finally, the placement layer is responsible for distributing a set of shards to the archives. Extraction is similar, but reversed; shards are requested from the archive, combined into fragments, and those fragments are combined into objects.

The two levels of secret splitting provide three important security advantages. First, as Figure 5 illustrates, two levels of splitting results in a tree, providing extra security through increased fan-out; even with all of the members of a shard tuple, an attacker can only rebuild a fragment, which provides no information about the shards for the other fragments. Second, as secret splitting algorithms present varied features, each split can be independently tuned for a specific property, and can select the algorithm best suited to that property.



(a) four data transformation layers in POTSHARDS

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

(b) inputs and outputs for each transformation layer

Fig. 5. The transformation component consists of four levels. Approximate pointers are utilized at the second secret split. Note that locating one shard tuple provides no information about locating the shards from other tuples.

Third, it enables recovery by allowing useful metadata to be stored with the fragments; this data will be kept secret by the second level of splitting.

4.2.1 *Preprocessing Layer.* When a user submits data to the POTSHARDS client for ingestion, objects are created from the user's files in a three-step process. First, each file is divided into a series of fixed-sized blocks. As the system is designed for archival workloads, these blocks are on the order of several hundred kilobytes to a megabyte in size. Second, as Figure 6 details, an object identifier is generated and appended to the block. Third, a hash over the block and id is generated and appended. This hash is used to confirm a successful rebuild during reads. It does not, however, compromise security, as it is included in the unconditionally secure secret split in the later stages of shard production.

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

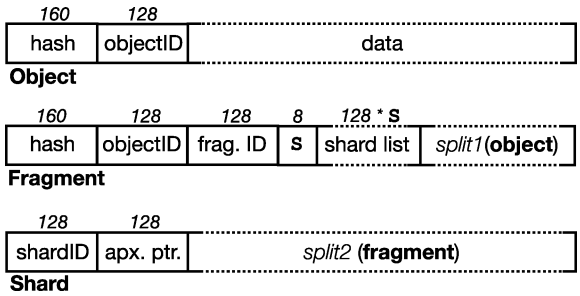


Fig. 6. Data entities in POTSHARDS, with size (in bits) indicated above each field. Note that entities are not shown to scale relative to one another. S is the number of shards that the fragment produces. split1 is an XOR secret split and split2 is a Shamir secret split in POTSHARDS.

As Figure 6 illustrates, each fragment contains metadata that assists in reconstruction and recovery. First, as in the object, a hash over the entire fragment serves to confirm a successful reconstruction. Second, the object identifier that this fragment contributes to aids in reconstruction; if a user is able to reproduce all of her fragments, this identifier assists in combining them into objects. This approach does not compromise security, as reconstructing a single fragment provides no information about which shards form the other fragments for a given object. Third, the fragment contains its own id. Finally, each fragment contains a list of the shards it produces.

A tuple of shards is produced from a fragment using another layer of secret splitting. This second split is tuned for availability, and therefore the current implementation of POTSHARDS uses an m of n secret splitting algorithm [Shamir 1979]. As mentioned in Section 3.1, this allows reconstitution in the event that an archive is down or unavailable when a request is made.

As Figure 6 shows, shards contain no information about the fragments that they make up. They do, however, include two pieces of metadata. First, they include their own shard id. Second, they include an approximate pointer to a random shard from the same shard tuple, as described in Section 4.1.

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

One drawback of the two-level secret splitting approach is the resulting increase in storage requirements. A two-way XOR split followed by a 2 of 3 secret split increases storage requirements by a factor of six; distributed RAID, and metadata further increase the overhead. If a user desires to offset this cost, data

can be submitted in a compressed archival form [You et al. 2005]; compressed data is handled just like any other type of data.

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

This layer contributes to security in four ways. First, since it is part of the data transformation component, no knowledge of which shards belong to an object need exist outside of the client. Second, the effectiveness of an insider attack at the archives is reduced because no single archive contains enough shards to reconstitute any data. Third, the effectiveness of an external attack is decreased because shards are distributed to multiple archives, each of which can exist in its own security domain. Fourth, the placement layer can take into account the geographic location of archives in order to maximize the availability of data.

4.3 Archive Design

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

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

As in other distributed RAID systems [Stonebraker and Schloss 1990; Chang et al. 2002], fault-tolerant, distributed storage is achieved by computing parity across unrelated data in wide-area redundancy groups. Given an (n, k) erasure code, a redundancy group is an ordered set of k data blocks and $n - k$ parity blocks where each block resides on one of n distinct archives. The redundancy group can survive the loss of up to $n - k$ archives with no data loss. The current implementation of POTSHARDS has the ability to use Reed-Solomon codes or single parity to provide flexible and space-efficient redundancy across the archives.

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

will result in a roll-back of the parity updates, and replacement of the shard into another block.

An integral part of preserving data, POTSHARDS actively verifies the integrity of data using two different forms of checking. First, each archive actively monitors the integrity of its own contents using stored hashes. Second, interarchive integrity checking is performed using algebraic signatures [Schwarz and Miller 2006] across the redundancy groups. Algebraic signatures have the property that the signatures of the parity equals the parity of the signatures. This property is used to verify that the archives in a given redundancy group are properly storing data and are performing the required internal checks.

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

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

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

Once the fail-over archive is selected, recovery occurs in multiple rounds. A single round of the secure recovery protocol is illustrated in Figure 7. In this example, the available members of a redundancy group collaborate to reconstruct the data from a failed archive onto a chosen archive, X. An archive, which cannot be the fail-over, is appointed to manage each round (in Figure 7, archive A has been selected). The managing archives determines the ordering for the round and generates a request containing an ordered list of archives,

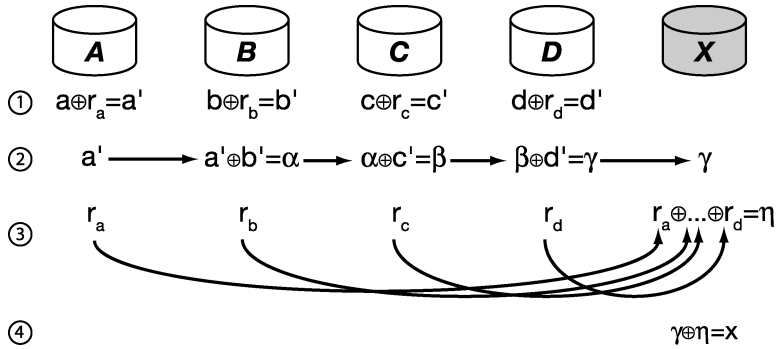


Fig. 7. A single round of archive recovery in a RAID 5 redundancy group. Each round consists of multiple steps. Archive N contains data n and generates random blocks r_n.

the id of the block to regenerate, and a data buffer. Each archives in the list then proceeds as follows.

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

This continues at each archive until the chain ends at archive X and the block is reconstructed. The commutativity the rebuild process allows decreases the likelihood of data exposure by permuting the order of the chain in each round. This procedure is easily parallelized and continues until all of the failed blocks for the redundancy group are reconstructed. This approach can be generalized to any linear erasure code; as long as the generator matrix for the code is known, the protocol remains unchanged.

4.4 User Indexes

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

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

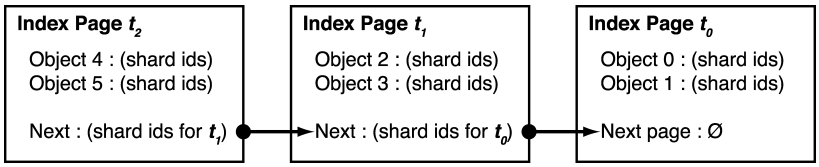


Fig. 8. User index stored in POTSHARDS as multiple pages. The initial page was created at time t_0, subsequent pages at times t_1 and t_2, respectively. By knowing just the shards to the newest page, the user can extract the entire index.

authentication information to sufficiently pose as the user) is assumed to have acquired full access to the user's data.

The index for each user can be stored in POTSHARDS as a linked list of index pages, with new pages inserted at the head of the list, as shown in Figure 8. Since the index pages are designed to be stored within POTSHARDS, each page is immutable. When a user submits a file to the system, a list of mappings from the file to its shards is returned. This data is recorded in a new index page, along with a list of shards corresponding to the previous head of the index list. This new page is then submitted to the system and the shard list returned is maintained as the new head of the index list. These index root-shards can be maintained by the client application or even on a physical token, such as a flash drive or smart-card.

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

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

4.5 Recovery with Approximate Pointers

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

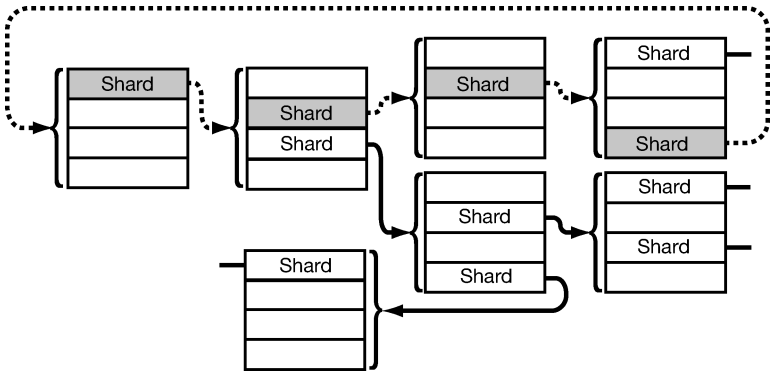


Fig. 9. Recovery example where each approximate pointer indicates a region of four shard names. If shards are produced using a 2 of 4 split, the ring heuristic reveals one recovery candidate based on its circular-linked list structure of *exactly* n, four, shards (shaded shards and dotted approximate pointers). In contrast, the naïve approach of testing paths of length m, 2, would result in many more potential recovery candidates.

With both reconstruction strategies, the process starts the same way. Once a user determines that she must recover her data, perhaps due to a lost index, she begins by collecting her shards. As Section 4.1 described, the user's shards can be identified by the initial, user id portion of the shard name. The operation to collect all of the shards could differ for each archive. Additionally, releasing all of a user's shards is a potentially dangerous; a lot of data could be compromised. Therefore, this operation should require a higher level of authorization and clearance.

In the first recovery strategy, the naïve approach, the solution space is reduced by limiting reconstruction attempts to paths of length m. This approach can be useful when less than the full set of shards are available; with less than a full set of shards, the user may not have all n shards that reconstruct a fragment. Unfortunately, a number of factors conspire to make this approach less than ideal. First, as Figure 9 illustrates, while still better than a purely brute force-based approach, there are still a fair number of paths of length m, and therefore many possible candidates for reconstruction. Second, a side effect of the randomization discussed in Section 4.1 is that reconstruction with less than a complete tuple of shards is time consuming; secret splitting is expensive, and a user with less than n shards does not have a total ordering, and must attempt recovery on multiple permutations. For example, suppose that a user posses a chain of three of the five shards, S_a, S_b, S_c, resulting from a 3 of 5 threshold split. If the inter-shards links were not formed using the randomization method, but rather were simply formed using the name order, reconstruction would potentially involve testing three shard tuples: $\langle S_a, S_b, S_c, \emptyset, \emptyset\rangle$, $\langle \emptyset, S_a, S_b, S_c, \emptyset\rangle$, and $\langle \emptyset, \emptyset, S_a, S_b, S_c\rangle$. However, if the shards were connected using the randomized method, reconstruction attempts would need to include combinations with interspersed empty shards, such as $\langle \emptyset, \emptyset, S_a, S_b, S_c\rangle$, $\langle \emptyset, S_a, \emptyset, S_b, S_c\rangle$, $\langle \emptyset, S_a, S_b, \emptyset, S_c\rangle$, ...

The second strategy, the ring heuristic, utilizes the circular structure of the shard tuples, depicted in Figure 9. This approach only attempts to reconstruct cycles of length n, and provides two important advantages. First, it efficiently reduces the potential solution space to a manageable number of recovery candidates. Second, because the ring heuristic identifies all n members of a shard tuple, the shard names impose a full ordering. One disadvantage of this approach is that it requires a lot of shards. However, given an incomplete set of shards, the ring heuristic can be used as a first-pass algorithm to quickly recover the full shard tuples and reduce the solution space for the remaining, unrecovered secrets. These can then be recovered using the naïve approach.

In addition to approximate pointers, there are other hints in the structure of the data entities, illustrated in Figure 6, that are useful with both the naïve approach and the ring heuristic. First, a hash of the fragment is used to confirm a successful reconstruction. Second, each reconstructed fragment includes a list of the secret shards that it produces. Using this list, reconstruction of a secret from less than n shards will reveal the ids of the $n-m$ shards that were not used. In a recovery scenario, the shards that correspond to these ids can be removed from the set of unused shards, thereby reducing the remaining solution space. It is important to note these hints are primarily useful *after* a block has been reconstructed; less than m of n shards contain *no* information, and the hints themselves are only present in the reconstructed block.

5. EXPERIMENTATION AND DISCUSSION

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

The current version of POTSHARDS consists of roughly 1,400 lines of Python version 2.5 code. For improved buffer management, versus standard Python lists, SciPy version 1.1.0 arrays were utilized extensively. Further, while most of the current version is implemented in native Python with SciPy code, an exception was the threshold secret splitting scheme. For this, we utilized an optimized C library that includes a $GF(2^8)$ arithmetic based implementation of Shamir's linear interpolation algorithm.

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

During these experiments, the data transformation component utilized block sizes of 750 KB. Since POTSHARDS is designed for archival storage, block sizes are expected to be relatively large, on the order of a few hundred kilobytes to a megabyte, and possibly larger. Additionally, the default approximate pointer width, R, was 30. Unless otherwise noted, the first layer of secret splitting

Table II. Ingestion and Extraction Performance for a Variety of Configurations

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

The splitting parameters are expressed in tuples of the form $(m, n,$ `algorithm` where the first tuple corresponds to the first split, and the second tuple to the second split. For testing, a pass-through algorithm named "null" was created which appends metadata but does no secret splitting.

used an XOR-based algorithm and produced two fragments per object, and the second layer utilized a 2 of 3 Shamir threshold scheme. The workloads consisted of randomly generated files, all larger than 1 MB in size. While these files are representative of the files that a long-term archive might contain, it is important to note that POTSHARDS sees all objects as the same, regardless of the object's origin or content.

5.1 Read and Write Performance

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

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

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

Finally, Table II shows the client throughput with a first-level XOR split and a second-level Shamir split, the "default" POTSHARDS configuration. This arrangement demonstrated the slowest throughput rates, although this is to be expected for a number of reasons. First, with two levels of secret splitting, there

are two levels incurring a random data generation penalty. Second, and more importantly, with an initial (2, 2, `xor`) split, followed by a (2, 3, `Shamir`) split, the second-level splitter is splitting over twice as much data as the user had submitted. Further in our experiments, system throughput is measured from the user's perspective; demands inside the system are six times those seen by the client.

5.2 Archive Reconstruction

The archive recovery mechanisms were run on our local system using eight 1.5 GB archives. Each redundancy group in the experiment contained eight archives encoded using RAID 5. A 25 MB client workload was ingested into the system using 2 of 2 XOR splitting and 2 of 3 Shamir splitting, resulting in 150 MB of client shards, excluding the appropriate parity. After the workload was ingested, an archive was failed. We then used a static recovery manager that sent reconstruction requests to all of the available archives and waited for successful responses from a fail-over archive. Once the procedure completed, the contents of the failed archive and the reconstructed archive were compared. This procedure was run three times, recovering at 14.5 MB/s, with the verification proving successful on each trial. The procedure was also run with faults injected into the recovery process to ensure that the verification process was correct.

5.3 User Data Recovery

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

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

With a baseline established, we evaluated usefulness of approximate pointers with both the naïve and the ring heuristic described in Section 4.5. While user indices provide for efficient read and write performance under most access scenarios, Figure 10 shows that approximate pointers can provide adequate recovery performance when an index is unavailable. As the number of shards increases, the ring heuristic provides dramatically faster recovery times when

Table III. Recovery Time, in Seconds

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

These recovery times are for a variety of secret splitting parameters using the brute-force approach in which approximate pointers are not used.

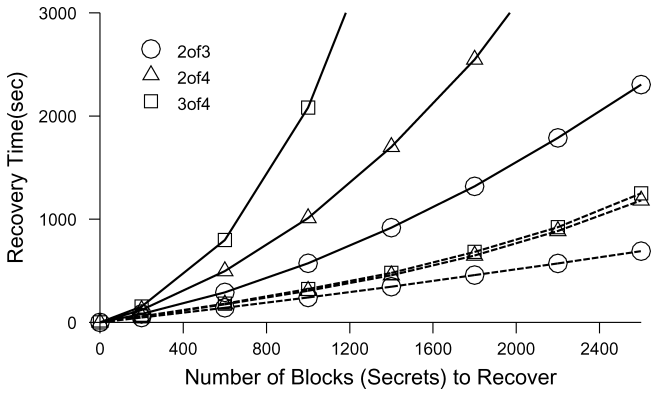
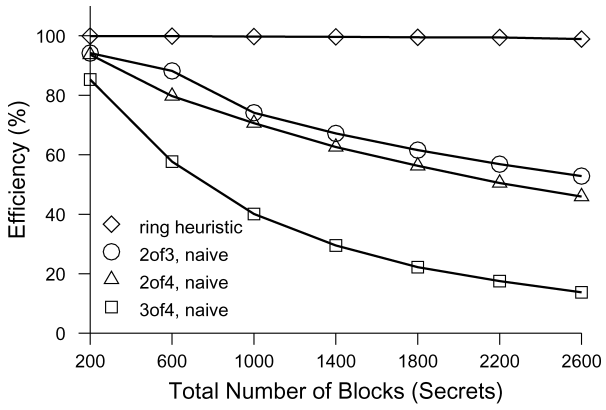


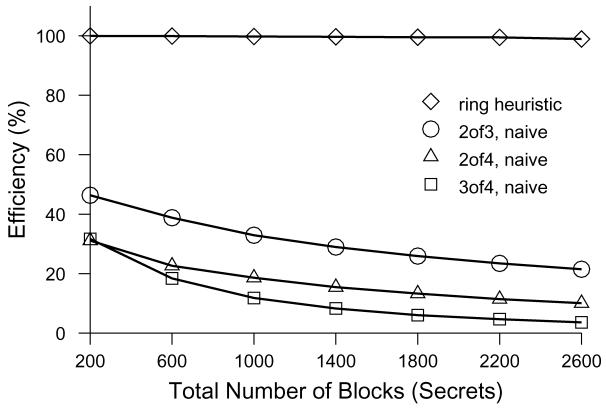
Fig. 10. Recovery time, in seconds, for various values of m and n with both the naïve approach and the ring heuristic. Reconstruction plots that use the ring heuristic are shown using a dashed line.

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

Recovery times are largely computationally limited because m of n threshold schemes often rely upon expensive operations. Thus, in addition to recovery times, we can also measure the efficiency of the strategies based on how often they select false shard tuples. By this definition, perfect efficiency would be achieved if every shard tuple selected reconstructed a valid secret. Figure 11(a) shows the comparison of three different secret splitting settings and their recovery efficiency. From our experiments, two things are evident. First, the ring heuristic is very efficient at selecting valid shard tuples with all three of the secret splitting settings. Second, larger values of m adversely affect the efficiency of the naïve approach. This is due to the fact that as m increases, the number of paths of length m increases greatly. Given a shard with an approximate pointer that points to R candidate shards, and a namespace density of $D = (0, 1]$, there are $(RD)^{m-1}$ possible paths. Thus, on average, $\frac{1}{2}(RD)^{m-1}$ paths must be tested by the naïve approach.



(a) Share Tuple Selection Efficiency



(b) Shamir Call Efficiency

Fig. 11. Efficiency of different recovery strategies as the total number of shards increases. Efficiency of shard tuple selection is the percentage of tuples selected by the recovery heuristic that reconstruct a valid secret. Efficiency of the Shamir call is the percentage of reconstruction attempts that reconstruct a valid secret. The ring heuristic was used with all three secret splitting parameter settings and each gave similar results. Thus, all of the results obtained using the ring heuristic were averaged and shown as one plot in order to improve clarity.

The difference between the ring heuristic and the naïve approach is even more pronounced when the efficiency of the secret splitting operation is measured. Figure 11(b) clearly illustrates two important points. First, the ring heuristic benefits from a full shard tuple and thus a total ordering over the secret shares. Therefore, each potential shard tuple selected by the ring heuristic only needs to be tested by the Shamir reconstruction operation once. Figure 11 shows the result; the efficiency of the ring heuristic is the same at the shard tuple selection level as it is at the secret splitting operation level. In contrast, with only m of the total n secret shares, the shard names only provide a partial ordering. Thus, a shard tuple selected by the naïve scheme must be tested by the secret splitting reconstruction operation up to $\frac{n!}{m!(n-m)!}$ times before it can be confirmed as invalid.

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

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

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

The population of an approximate pointer also has an effect on data recovery times. Even with the ring heuristic, recovering objects from shards, when faced with no other outside information, amounts to controlling a combinatorics problem of exponential growth. This is evident in Figure 12 which shows the recovery time for 2,600 secrets at various population levels per pointer. Population was

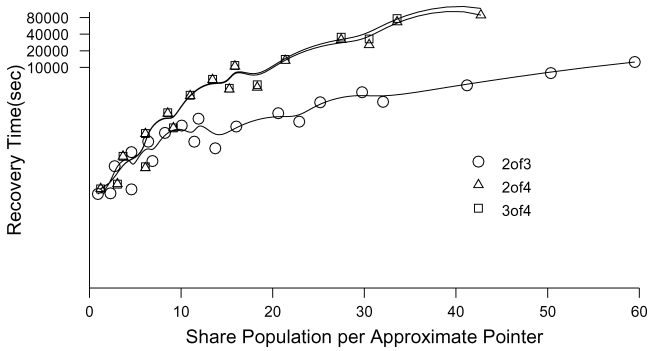


Fig. 12. The effect of the approximate pointers' populations on the time to recover 2,600 secrets using the ring heuristic. In these tests, population, P, was modified by adjusting the size of the region, R, indicated by the approximate pointer; density was kept constant.

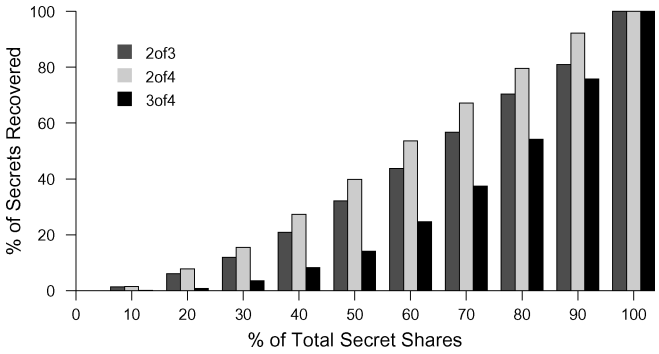


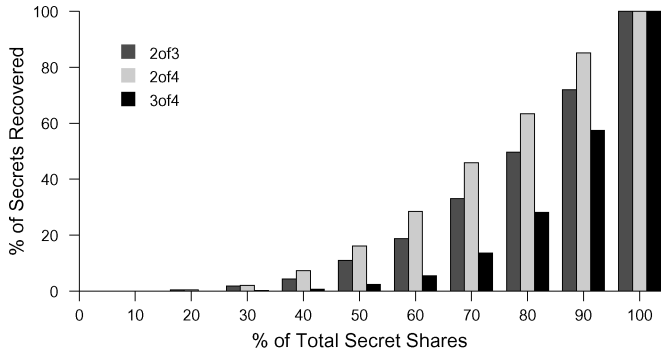
Fig. 13. Percentage of 2,600 total blocks (secrets) that could be recovered by an adversary in a large-scale compromise. Tests were performed over randomly selected, partial sets of secret shares.

increased by modifying the width, R, of the approximate pointers; the shard density was constant. The tests were run utilizing the ring heuristic and, as would be expected, the tests that required cycles of length four to be tested grew at a faster rate than those that only had to test cycles of length three.

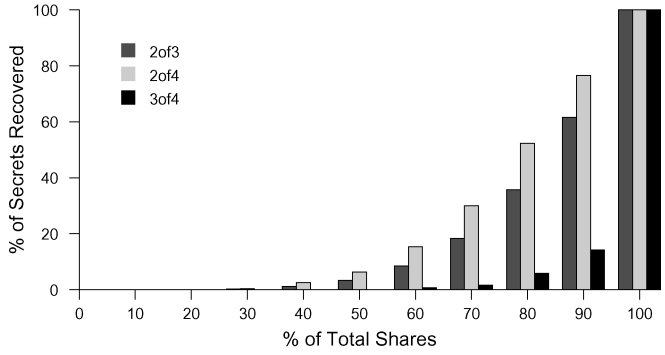
5.4 Security Model

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

Higher values of m provide a higher level of data protection, but can also lead to higher recovery times. As Figure 13 and Figure 14 illustrate, less data was leaked when larger values of m were used. However, there is the risk that recovery times will be higher if less than the full shard tuple can be acquired. While approximate pointers and the naïve approach are still useful, Figure 10 and Figure 11 demonstrate that higher values of m incur a penalty for recoveries



(a) Top Level XOR Split of Two



(b) Top Level XOR Split of Three

Fig. 14. Percentage of 1,300 total blocks (secrets) that could be recovered by an adversary in a large-scale compromise. Tests were performed over randomly selected, partial sets of secret shares in which secrets were guarded through two levels of secret splitting: a top-level XOR split and a lower-level threshold split.

with less than a full set of n shards. This scenario can, however, be mitigated in two ways. First, a hybrid solution can be utilized in which as many secrets as possible are recovered using the ring heuristic. Then, the remaining shards can be recovered using the naïve approach. Second, as Figure 6 illustrates, the list of shard identifiers that a fragment generates is appended to the fragment. Thus, upon successful reconstruction of a fragment from only m shards, the remaining shards can be identified and removed from the list of unused shards. This reduces the solution space for the subsequent secret recoveries. The results of our experiments suggest that larger values of m should be chosen when secrecy is a priority over potential recovery times.

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

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

Despite their increased namespace overhead, higher values of n do provide security benefits. As Figure 13 shows, higher values of n can be useful for limiting the amount of information leaked, albeit mostly as a result of allowing higher values of m. To this end, our experimentation suggests that a higher value of n, along with a correspondingly higher value of m, provides the most protection against lost data.

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

To determine the amount of data that can be revealed from a large-scale compromise, and to better understand how to limit it, we measured the data that could be regenerated from a random subset of secret shares. In our experiment, 2,600 secrets were split using Shamir's linear interpolation scheme. From the resulting set of secret shares, an increasing percentage was randomly selected and as much data as possible was recovered. The results, shown in Figure 13, indicate two things. First, less data is released when both m and n increase and $n - m$ is held constant. In our experiment, using a 3 of 4 split revealed the least amount of information. Second, for a fixed m, increasing n reveals an increasing amount of information. This is not unexpected as the odds of randomly selecting a secret share from a given tuple increase as the size of the tuple increases. In fact, threshold schemes are often used because of the availability that can be achieved by increasing the value of n.

One approach to minimizing data loss from large-scale compromises is the two-level secret splitting technique used by POTSHARDS. To test the benefits of this strategy, we utilized an initial n of n XOR-based split. Each of the resulting shares is then split using Shamir's m of n threshold scheme. The results, for two different values of n at the XOR split, are shown in Figure 14 and indicate that the additional level of secret splitting is effective at lowering the amount of information released. Also, as in the single split, larger values of $n - m$ at the second layer of secret splitting still resulted in higher amounts of information loss.

Our experiments also indicate that a larger split at the first level of splitting further limited the amount of information loss. This is evident in Figure 14(b), which shows that revealing 20% of the total number of secret shares under a first-level 3 of 3 split revealed no data, regardless of the second-level split. The same 20% compromise with a first-level split of two and second-level 2 of 3 or 2 of 4 split resulted in 0.42% and 0.50% of the total number of secrets being revealed, respectively. Even with 60% of the total number of secret shares and a

3 of 4 Shamir split, a first-level split of three only revealed an average of 0.68% of the secrets. In contrast, with 60% of the secret shares, a first-level split of two revealed 5.46% and the single layer of splitting alone revealed 24.69%. Of course, 60% of the total number of secret shares represents a very large-scale compromise—over half of the shares stored for the user have been acquired; we expect that compromises are more likely to result in 10% or fewer shares being acquired, given the intrusion detection approaches made possible by sparse namespaces.

Of course, while our results do show that multiple layers of secret splitting enhance security, they do incur a storage penalty. As Figure 6 shows, there is already a constant amount of storage overhead in the form of hashes and identifiers. These costs are, however, dominated by the storage blowup intrinsic to secret splitting. This situation is exacerbated by multiple levels of splitting. For example, a first-level split of three, along with a second 3 of 4 split incurs a storage blowup of twelve.

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

5.4.3 *Chaff Shards.* When a shard that does not exist is requested, either mistakenly or due to a malicious user, there are two possible responses: an error message or a chaff shard. The use of chaff [Bellare and Boldyreva 2000; Rivest 1998] (fake packets) has been suggested as an approach to providing data secrecy without encryption. A key difference, however, is that the "chaffing and winnowing" strategy uses chaff as its primary secrecy mechanism. In the model that we are investigating, secrecy comes primarily from secret splitting. Thus, in our model, when a request is made for a shard that does not exist, a seemingly valid chaff shard is generated and returned to the user.

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

There are two possible strategies for dealing with a user that requests a shard multiple times in order to test its validity; if a shard is requested twice, but the

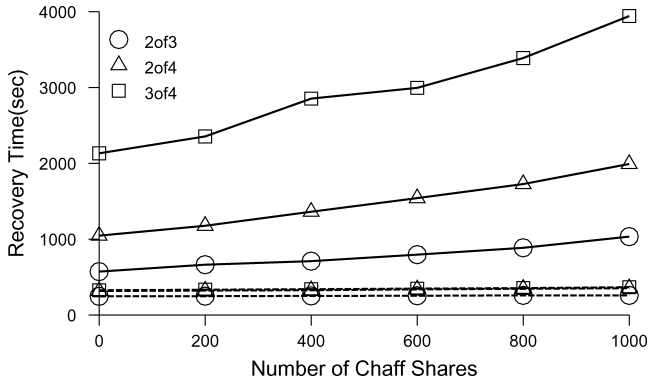


Fig. 15. The effect of chaff on the time to recovery 1,000 secrets. Recovery attempts that utilized the ring heuristic are shown using a dashed line.

returned result is different each time, it is clear to the user (or attacker) that the shard is simply chaff. First, chaff can be generated using a deterministic process. Alternatively, the chaff can be generated randomly and then stored. One issue with the second strategy is that a user could attempt to intentionally request false shards in order to pollute a user's set of shards. The aim of such an attack might be to use the increased recovery time as a type of denial-of-service attack. In fact, as Figure 15 demonstrates, chaff does not dramatically increase the recovery time, especially if the user is able to utilize the ring heuristic.

6. FUTURE WORK

Currently, POTSHARDS depends on strong authentication and intrusion detection to keep data safe, but it is not clear how to defend against intrusions that may occur over many years, even if such attacks are detected. We are exploring approaches that can refactor the data [Wong et al. 2002] 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 reencryption, refactoring for security could be done over time to limit the window over which a slow attack could succeed. Refactoring could also be applicable to secure migration of data to new storage devices.

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. Others have begun to address this problem [Gladney and Lorie 2005], 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. CONCLUSION

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

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 reconstitute a single fragment of user data; because he does not know which particular shares are needed, he must obtain *all* of the possibly required shares. Second, we demonstrated that a user's data can be rebuilt in a relatively short time from the stored shards *only* if sufficiently many pieces can be acquired. Even a sizable (but incomplete) fraction of the stored pieces from a subset of the archives will not leak information, ensuring that data stored in POTSHARDS will remain secret. Third, with approximate pointers and a sparse namespace, 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.

Out experiences with an early implementation show that users can store data at over 4 MB/s and retrieve user data over 6 MB/s. Since POTSHARDS is an archival storage system, throughput is more of a concern than latency, and even these unoptimized throughputs exceed typical long-term data creation rates for most environments. Since the storage process is parallelizable, additional clients increase throughput until the archives' maximum throughput is reached; similarly, additional archives linearly increase maximum system throughput.

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

By addressing the long term threats to archival data while providing reasonable performance, POTSHARDS provides reliable data protection specifically

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

ACKNOWLEDGMENTS

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this article.

REFERENCES

104TH Congress. 1996. Health Information Portability and Accountability Act. http://www.hhs.gov/ocr/hipaa/.

ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. 2006. A fresh look at the reliability of long term digital storage. In *Proceedings of EuroSys*, 221–234.

BELLARE, M. AND BOLDYREVA, A. 2000. The security of chaffing and winnowing. In *Proceedings of the Advances in Cryptology 6th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'00)*. Lecure Notes in Computer Science, vol. 1976, Springer, Berlin, 517–530.

BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. 2003. Address obfuscation: An effcient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 105–120.

CHANG, F., JI, M., LEUNG, S.-T. A., MACCORMICK, J., PERL, S. E., AND ZHANG, L. 2002. Myriad: Cost-Effective disaster tolerance. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. 103–116.

CHOI, S. J., YOUN, H. Y., AND LEE, B. K. 2003. An efficient dispersal and encryption scheme for secure distributed information storage. Lecture Notes in Computer Science, vol. 2660, 958–967.

CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. 2001. Freenet: A distributed anonymous information storage and retrieval system. Lecture Notes in Computer Science, vol. 2009, 46–66.

CLEVERSAFE. 2006. Highly secure, highly reliable, open source storage solution. http://www. cleversafe.org/.

FORREST, S., SOMAYAJI, A., AND ACKLEY, D. H. 1997. Building diverse systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 67–72.

GLADNEY, H. M. AND LORIE, R. A. 2005. Trustworthy 100-year digital objects: Durable encoding for when it's too late to ask. *ACM Trans. Inf. Syst. 23,* 3, 299–324.

GOLDBERG, A. V. AND YIANILOS, P. N. 1998. Towards an archival intermemory. In *Proceedings of the Conference on Advances in Digital Libraries (ADL'98)*, 1–9.

GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networking (DSN'04)*.

GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. 2005. Deconstructing commodity storage clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 60–71.

HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. 2005. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

HAND, S. AND ROSCOE, T. 2002. Mnemosyne: Peer-to-peer steganographic storage. Lecture Notes in Computer Science, vol. 2429, 130–140.

IYENGAR, A., CAHN, R., GARAY, J. A., AND JUTLA, C. 1998. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC'98)*, 123–135.

KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. 2003. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 29–42.

KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. 2004. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*.

KING, S. T. AND CHEN, P. M. 2003. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 223–236.

KOTLA, R., ALVISI, L., AND DAHLIN, M. 2007. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference*, 129–142.

MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. 2005. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst. 23,* 1, 2–50.

MILLER, E. L., LONG, D. D. E., FREEMAN, W. E., AND REED, B. C. 2002. Strong security for network-attached storage. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, 1–13.

OXLEY, M. G. 2002. (H.R.3763) Sarbanes-Oxley Act of 2002.

PLANK, J. S. 1997. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Softw. Practice Exper. 27,* 9, 995–1012. Correction in James S. Plank and Ying Ding, Tech. rep. UT-CS-03-504, University of Tennessee, 2003.

QUINLAN, S. AND DORWARD, S. 2002. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. USENIX. 89–101.

RABIN, M. O. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM 36*, 335–348.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: The OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 1–14.

RIVEST, R. L. 1998. Chaffing and winnowing: Confidentiality without encryption. CryptoBytes, *4*, 1, 12–17.

SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, 110–123.

SCHWARZ, S. J., T. AND MILLER, E. L. 2006. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the IEEE 26th International Conference on Distributed Computing Systems (ICDCS'06)*.

SHAMIR, A. 1979. How to share a secret. *Comm. ACM 22,* 11, 612–613.

STINSON, D. R. 2002. *Cryptography: Theory and Practive*, 2nd ed. The CRC Press Series on Discrete Mathematics and its Applications. Chapman and Hall (CRC), Boca Raton, FL.

STONEBRAKER, M. AND SCHLOSS, G. A. 1990. Distributed RAID—A new multiple copy algorithm. In *Proceedings of the 6th International Conference on Data Engineering (ICDE'90)*, 430–437.

STORER, M., GREENAN, K., MILLER, E. L., AND MALTZAHN, C. 2005. Potshards: Storing data for the long term without encryption. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*.

STORER, M. W., GREENAN, K. M., AND MILLER, E. L. 2006. Long Term threats to secure archives. In *Proceedings of the ACM Workshop on Storage Security and Survivability*.

STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. 2007. POTSHARDS: Secure long-term storage without encryption. In *Proceedings of the USENIX Annual Technical Conference*, 143–156.

SUBBIAH, A. AND BLOUGH, D. M. 2005. An approach for fault tolerant and secure data storage in collaborative work environements. In *Proceedings of the ACM Workshop on Storage Security and Survivability*, 84–93.

TRONCOSO, C., DE COCK, D., AND PRENEEL, B. 2008. Improving secure long term archival of digitally signed documents. In *Proceedings of the ACM Workshop on Storage Security and Survivability*, 27–36.

WALDMAN, M., RUBIN, A. D., AND CRANOR, L. F. 2000. Publius: A robust, tamper-evident, censorship-resistant Web publishing system. In *Proceedings of the 9th USENIX Security Symposium*.

WANG, X., LI, Z., XU, J., REITER, M. K., KIL, C., AND CHOI, J. Y. 2006. Packet vaccine: Black-Box exploit detection and signature generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*.

WONG, T. M., WANG, C., AND WING, J. M. 2002. Verifiable secret redistribution for threshold sharing schemes. Tech. rep. CMU-CS-02-114-R, Carnegie Mellon University. October.

WYLIE, J. J., BIGRIGG, M. W., STRUNK, J. D., GANGER, G. R., KILIÇÇÖTE, H., AND KHOSLA, P. K. 2000. Survivable storage systems. *IEEE Comput.*, 61–68.

XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. 2005. Automatic diagnosis and responses to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, 223–234.

YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. 2005. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*.

ZHANG, Z., LIAN, Q., LIN, S., CHEN, W., CHEN, Y., AND JIN, C. 2007. BitVault: A highly reliable distributed data retention platform. *ACM SIGOPS Oper. Syst. Rev. 41,* 2, 27–36.

Received September 2008; revised January 2009; accepted February 2009