PERCIVAL: A RELIABLE, LONG-TERM, DISTRIBUTED STORAGE SYSTEM FREE OF FIXED-KEY ENCRYPTION

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Joel Cameron Frank

June 2017

The Dissertation of Joel Cameron Frank is approved:

Professor Darrell D.E. Long, Chair

Professor Ethan L. Miller

Professor Ahmed Amer

Dean Tyrus Miller
Vice Provost and Dean of Graduate Studies
Table of Contents

List of Figures v
List of Tables viii
Abstract ix
Dedication x
Acknowledgments xi

1 Introduction 1
  1.1 Determining System Requirements 2
  1.2 Searching in an Untrusted Environment 3
  1.3 Disaster Recovery 5

2 Background 7
  2.1 Secret Splitting 7
  2.2 Access Control 8

3 Related Work 10
  3.1 Trace Studies 10
    3.1.1 Enterprise Studies 10
    3.1.2 Long-Term Storage Systems 11
  3.2 Long-Term Storage 11
  3.3 Searching Over Encrypted Data 14
  3.4 Search Space Reduction 15

4 Evolutionary Trends in a Supercomputing Tertiary Storage Environment 17
  4.1 Methodology 17
    4.1.1 Evolution of the NCAR Mass Storage System 18
    4.1.2 Archive Storage Traces 19
  4.2 Results 20
    4.2.1 Read Density 21
    4.2.2 Inter-Reference Intervals 22
    4.2.3 Latency to First Byte 23
    4.2.4 Hourly Usage Patterns 25
    4.2.5 Weekly Usage Patterns 26
    4.2.6 File Sizes 27
    4.2.7 Directory Density 30
List of Figures

4.1 Memory and storage hierarchy in large computer systems. This is also known as the storage pyramid. .................................................. 18
4.2 Comparison of read, write, and overall inter-reference intervals for the 1992 and 2010 trace data. It is evident that only 30% of the 2010 archive is being accessed as compared to 80% in 1992. (Note: In this figure, 100% is the total number of files in the system, 69 million, not the total number of files seen in the traces.) .............. 23
4.3 Comparison of latencies to first byte. All latencies have improved with the exception of primary tape, whose latency worsened by a factor of 1.5 ................... 24
4.4 Comparison of the average amount of data transferred per hour to the number of read and write events per hour. 0 is Sunday at midnight. ......................... 26
4.5 Data transferred per week in the two archives. In 1992, overall workload increased over the trace period, but write rate did not. In 2010, there was a relatively constant workload across the trace period, including holidays, which implies an increase in the use of batch processing. .................................................. 28
4.6 File size comparisons in the two archives. The max size in the 1992 archive was 200 MB due to the physical limit of the storage media at the time. Most of the space in 2010 is consumed by large files, even though most of the files are small. .......... 29
4.7 Comparison of directory density between the 1992 and 2010 data sets. The number of files per directory has roughly doubled over the 20 year period. .............. 30
5.1 Comparison of two possible configurations. ........................................ 34
5.2 Overview of information flow during a query. (1) A client starts the process by sending a search request to each of the query servers. (2) Each query server responds with the correct share of a secret-split reverse index, or a non-empty response signifying ‘not found’. (3) The client can then either make additional queries, or proceed and request the data of interest from the data servers by sending the appropriate identifier(s). (4) Each data server then responds with its share of the secret-split data object. ....... 35
5.3 Generating reverse indexes. Once the keywords for each data object have been identified, they are transformed into reverse indexes mapping keywords to their data object(s). .................................................. 37
5.4 Overview of ingesting a query share. The keyword and RBAC credential are sent to the HSM, while the query share is sent directly to the key-value store. Once the keyword has been processed into a hash string, the resulting hash and query share are stored in the server’s key-value store. ........................................ 37
5.5 Simple single search term query. A client starts the process by hashing the search
term with its RBAC credential. The resulting hash string is sent to each query server’s
HSM to be further hashed with that query server’s salt, and ultimately used to look
up the requested share of the secret-split reverse index. Once the client has received
the query shares, it is able to reconstruct the original reverse index: mapping the
search term to the appropriate data object identifier(s).  

5.6 CDF of Reverse Index Size. For each number of keywords selected per data object,
2/3 were found to contain a single data object identifier, and almost all reverse indexes
were found to contain fewer than ten identifiers. Since all shares must be the same
size, these would-be small shares need to be padded in order to make them the same
size as the largest shares, otherwise it enables targeted theft of those shares.  

6.1 Given a file size of 1 MB, a reconstruction threshold of three, $T = 3$, and a recon-
struction rate of 20 GB/sec, reconstruction would take approximately 1.5 years to
reconstruct 10,000 shares, as seen in the blow up detail of the lower left section of the
graph. 

6.2 Given a maximum allowed value of seven, $M = 8$, the random set, $S_0$, is randomly
generated for a single group of sibling shares. From this set, the sets $P_{01} - P_{0N}$ are
randomly generated and are stored with their respective share. 

6.3 Taking the union of two candidate share’s $P$ sets when $|S| = 4$. This is an example
of two candidate shares that cannot be siblings since here: $|P_{01} \cup P_{12}| > |S|$. 

6.4 Taking the union of two candidate share’s $P$ sets. This is an example of two candidate
shares that might be siblings since $|P_{01} \cup P_{12}| \leq |S|$. 

6.5 Performance comparison with no pre-filtering during reconstruction and pre-filtering
on pairs of values present in each share’s $P$ set. It can be seen in 6.5a, when pre-
filtering is disabled and $|S| \leq 8$, insufficient information is stored with each share to
allow the Set-Subset method to correctly identify all groups of sibling shares. 

6.6 The Set-Subset method implemented and validated against the theoretical model.
The black solid line denotes the mean performance with the sample population fitting
a $t$-distribution. Increasing the size of $S$ slightly decreases performance with the
benefit of improving resistance to loose targeted theft. The increase in required tests
when $|S| = 4$ and there are more than $10^9$ shares per server is due to decreased effect
that pre-filtering has at that $R_{sm}$. 

6.8 The Set-Subset method roughly performs between $O(n)$ and $O(n^2)$ depending on the
chosen values of $|S|$ and $M$. As one moves left on the curve, performance increases
at the cost of decreased resistance to loose targeted theft. The shaded areas denote
regions where the percentage of homogeneity is potentially low enough to enable loose
targeted theft depending requirements of the operating environment. 

6.9 Hint generation example: the hint, $H_1$, is generated by passing the share’s ID through
a hash function, and then extracting a predefined range of bits from the resulting hash
string. 

6.10 Example of generating hint shares by secret splitting the hint $H_1$. Each hint share,
$hs_{ij}$, is the hint share generated from the hint $H_i$ to be stored with share $j$ within
the group of sibling shares. 

6.11 A set of $N$ sibling shares, each with the appropriate set of hint shares, $hs_{ij}$, where
$hs_{ij}$ is the hint for share $i$ to be stored with share $j$. 

6.12 First step example: reconstruct the set of hint shares from the original hint, $H_1$
and this share’s hint share, $hs_{11}$. This allows the algorithm to pre-filter out those
candidate shares from server 2 whose hint share $hs_{12}$ does not match the required
value. Note: in this context, + refers to secret-split reconstruction.
6.13 The seven test reconstructions performed when considering three candidate shares, assuming a hint splitting threshold of two. Note: in this context, + refers to secret-split reconstruction.

6.14 Reconstruction tests required for an 8 bit hint, \( b = 8 \), and varying share counts per server. Increasing the splitting threshold improves security at the cost of increasing the number of reconstruction tests that need to be conducted due to the larger reconstruction space.

6.15 Example showing the state when the degrees of freedom afforded by the hint are greater than the number of shares per server. In this example, this occurs at \( 10^{9.6} \approx 2^{32} \).

6.16 The Secret-Split Hash method implemented and validated against the theoretical model. The black solid line denotes the mean performance with the sample population fitting a \( t \)-distribution.

6.17 Work required for varying hint sizes on a datastore with 1,000,000 shares per server. Performance can be maximized by choosing a hint size for the expected maximum size of the datastore, which will result in the optimal, i.e. linear, runtime. In this example, this occurs when \( b = 20 \).

7.1 Comparison of the average reconstruction throughput between the Crypto++ and JErasure libraries during 1000 tests. In general, the JErasure library performed three orders of magnitude faster than the Crypto++ library.
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Information of interest contained in a single trace record.</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Overall trace statistics, with activity normalized to an annual basis.</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of commercially available hardware security modules [48].</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Relative comparison of the key features between the most recent method, Approximate Pointers, and our proposed methods to prevent data loss in a secret-split data-store. Overall, both Set-Subset and Secret-Split Hash methods perform better than Approximate Pointers. These tests were run on a 4-core, 64 bit Linux machine with 24 GB of RAM.</td>
<td>52</td>
</tr>
</tbody>
</table>
Abstract

Percival: A Reliable, Long-term, Distributed Storage System Free of Fixed-key Encryption

by

Joel Cameron Frank

Secret splitting has been shown to improve reliability, reduce the risk of insider threat, and remove the issues surrounding key management in distributed long-term datastores. However, to date there has been little or no adoption of this technique in production environments. When it has been implemented, it was done relying on fixed-key encryption for various parts of the system, e.g. during ingestion to maintain user privacy, or pre-indexing to facilitate searching since the inherent security of such a datastore normally precludes it from being directly searched without reassembling the data. Fixed-key encryption, unfortunately, is not well suited for long-term applications due to its introduction of a single point of compromise and failure as well as its key management issues. Furthermore, even if the data remains intact after a long period of time, since standard reconstruction methodologies rely upon external knowledge to perform the reconstruction, they will eventually fail. When they do, information loss is almost certain in applications of sufficient size to make reconstruction combinatorially prohibitive. The most recent method to mitigate this risk has a high runtime, and limits the inherent security of the secret-split datastore.

To address the need of a reliable, long-term, distributed storage system free of fixed-key encryption, we propose Percival: a novel system that enables searching a secret-split datastore, maintains information privacy, and does not rely on external information to ensure reconstruction remains feasible. It is built upon the knowledge gained from conducting an in-depth comparison of file migration activity on the mass storage system (MSS) at the National Center for Atmospheric Research (NCAR) during two periods, one in the early 1990s, and another nearly twenty years later. To accommodate real-world user access patterns, Percival allows one to search the secret-split data while both keeping the bulk of the work on each client and the data custodians blinded to both the contents of a query as well as its results. Furthermore, to ensure reconstruction is feasible for even very large secret-split datastores, we also present two novel disaster recovery methods that greatly reduce the number of reconstruction attempts required during reconstruction; this enables recovery of the original data, where previously the data would have been lost.
To my parents, who taught me the only limits in life
are the ones we place on ourselves.

To my Uncle, who taught me there are no problems in life...only challenges.

And to my wife, Shayna, without whom none of this would have been possible.
Acknowledgments

I would like to thank both the members of my advancement committee and dissertation committee. To my advisor, Darrell Long, who always believed in me. To Ethan Miller, who taught me the devil is truly in the details. To Ahmed Amer, who was the balancing voice of reason amidst the storm. To Thomas Kroeger, who helped me find my path. Last but not least, Mark Stamp, who was an inspiration and fanned my passion for teaching.

In addition to my committee members, I would like thank my labmates. I could not possibly list all the things that I learned from those who came before. The lab truly was less once you moved on.

I would also like to thank my friends and family who were an ever present voice of encouragement and support.
Chapter 1

Introduction

In order to meet data availability and reliability requirements, storage system designers often turn to a distributed type architecture. However, when faced with the reality of insider threat, these types of systems often have difficulty meeting the security and information privacy requirements of their users. This gives rise for the need of a distributed storage system that is able to both meet the stringent performance requirements of its users while also providing provable security guarantees, even when the system has been compromised.

A secret-split datastore is a distributed storage system in which each geographically and administratively isolated server within the system stores secret-split data, i.e. shares, such that no server contains two shares from the same original data. Such datastores have been shown to improve both reliability and availability, as well as reduce the risk of insider threat in long-term storage applications.

The main attack vector of concern for a secret-split datastore is targeted theft. In general, targeted theft refers to an attacker identifying a small number shares by some characteristic and subsequently gaining access to only those shares. The typical targeted theft attack consists of an insider at a single location who has full access to the shares at that location, authorized access or not, and is attempting to reconstruct some data of interest by gaining access to the constituent shares on another server without detection. Furthermore, because the original data can be reconstructed if enough of the sibling shares are recombined, targeted theft typically results in the unauthorized release of information.

We define two types of this attack: strict and loose targeted theft. Strict targeted theft refers to an attacker identifying and accessing a small number of shares without detection using only a server’s public interface. For example, using POTSHARD’s public interface to query for a share based on its ID, but not being able to browse the collection of shares in any way. Building
upon this attack vector, *loose targeted theft* refers to an attacker not only using a server’s public interface, but also having the ability to access a small number of shares directly without detection.

Since it takes into account insider threat, loose targeted theft is a more realistic attack vector. It is assumed that such an attacker would be limited by standard monitoring practices, and as a result be able to only access small sections of the datastore at a time. The term ‘small’ is subjective, of course, and varies with system requirements and design. Neither of these methods allow an attacker to browse a server’s full contents since many other attacks can be performed once an attacker has achieved this level of compromise [79]. It is by these two definitions of targeted theft that we acknowledge that compromising a server is not a binary, i.e. an all-or-nothing action.

### 1.1 Determining System Requirements

Digital data storage is becoming increasingly important as the mechanism for transmitting scientific and cultural knowledge to future generations. As a result, organizations are collectively spending billions of dollars on systems that can preserve data for the long-term, yet there is surprisingly little known about how users actually *use* the data stored in archives—knowledge that could help system designers build better archives. Perhaps more importantly, there has been no research on how a long-term storage system *evolves* over relatively long periods of time. The study done by Agrawal, *et al.* at Microsoft [4] is one of the longest storage system studies conducted, and it covers only 5 years of desktop file systems, which is a relatively short time for long-term data. As a result, long-term storage system designers must rely more on “common sense” than on actual data.

To address this problem, we analyzed trace data from the mass storage system (MSS) at the National Center for Atmospheric Research (NCAR) for the period from 2008 to 2010 to determine data access frequencies, overall archive usage, response latencies, and trends in file size and density. We then performed the same analyses on the corpus of data gathered on the MSS at NCAR by Miller and Katz in 1990–1992 [52] in an effort to determine not only modern archival behavior, but evolutionary trends as well. Observations about long-term evolution are possible because NCAR’s primary mission—modeling the Earth’s climate—has remained relatively unchanged over the twenty year period covered by the two traces. Thus, these traces afford a unique opportunity not only due to the rarity of access to long-term storage traces, but also because they are both from the same long-term storage system separated by two decades. To our knowledge, this is the first time that the same system has been traced twice with nearly a two decade separation between the traces.

Our analyses conducted in this study yielded several surprising findings, in addition to confirming widely-held beliefs about scientific archival storage. Primary among these findings was a
dramatic shift in the read/write ratio from a read-dominated workload in 1992 to a write-dominated workload in 2010, a shift in the read/write ratio by a factor of four. This shift has major implications for archival storage system design, e.g. designing the system to primarily handle the write workload. Another key finding was that the fraction of the archive that was accessed more than once dropped from 80% in 1992 to 30% in 2010; a system with a lower access density may need to favor low storage cost over the ability to access files quickly, and may need more frequent archive scrubbing to compensate for the lack of user-driven accesses that may catch “bit rot”. The shift towards a write-dominated archive and decrease in the fraction of the datastore actually accessed suggest that future long-term storage systems may need to increasingly focus on preservation rather than providing high-speed access to long-term data. In addition, these two trends highlight the need for effective long-term data organization and search across millions to billions of files to identify the few files that are needed in response to a given query.

1.2 Searching in an Untrusted Environment

Security is often a critical issue for long-term storage, particularly given recent incidents involving insiders releasing large amounts of private or classified information [83]. Much of this risk is due to traditional storage systems having a single point of compromise: the data server. If that one point is compromised at any time during the datastore’s lifespan, information can be leaked. This threat is obviously magnified in a distributed environment since, by its nature, the data is stored in multiple locations. In situations where a single location, or site, is not trusted, but the collection of sites as a whole is trusted, secret splitting mitigates this problem. By first dividing a data object into shares, and then distributing each share to an independent site in the distributed environment, no single site has enough information to perform reconstruction because a single share reveals nothing about the original data.

Due to the inherent information-theoretic security of such a system, searching it is normally not possible without reconstructing the original data from its constituent shares. Repeated reconstruction, however, is not only computationally infeasible, it reintroduces the single point of compromise. As a result, the shares need to be pre-indexed in some way that facilitates searching them. Previously, this has been accomplished using fixed-key encryption, e.g. public-key, to minimize, and ideally prevent, information leakage. However, in addition to key management issues, which are undesirable in long-term storage environments, fixed-key encryption typically suffers from a catastrophic release of information upon compromise.

To address the need to maintain information privacy while searching a secret-split data-
store, we developed Percival: a novel system that accomplishes these tasks without relying on fixed-key encryption. Furthermore, Percival is completely agnostic with regards to the datastore’s implementation since whether the datastore is based on POTSHARDS [88] or Cleversafe [17], the user is left with an identifier that can be used to retrieve the user’s data. Percival combines this collection of identifiers with each data object’s search term(s) in order to produce a set of reverse indexes; each reverse index is in essence a search result since it maps a search term to the set of data objects that should be found using that search term. For our purposes, we define a search term as a single word that has been identified to relate to a particular data object. For example, to find the data object *Moby Dick*, one might use the search term ‘whale’ in order to retrieve the object from the datastore.

Once the set of reverse indexes is generated, each individual index is secret split; these resulting shares are each sent to a different query server in the distributed environment. We define a query server as a hardware security module backed by one or more machines working together as a single, logical key-value store. A hardware security module [55], HSM, is a commercially available, physical device that protects and manages the secure pieces of this design by providing a place to handle sensitive data in a relatively non-secure location. It provides both tamper evidence and resistance by logging intrusion attempts as well as clearing its internal memory if it detects an intrusion attempt. Percival relies on the HSMs to process all secure messages from the client, while not exposing any information to the rest of the query server. In general, query servers and their interaction to clients are discussed in detail in Section 5.2, but for now they can be viewed as a secure key-value store whose job it is to service search request from authorized clients, and respond with the share of the correct reverse index. Once a client has retrieved the shares to a single reverse index, it is able to reconstruct that index, thereby obtaining the set of data object identifiers that can then be used to retrieve the desired data from the underlying datastore.

Percival also provides mechanisms for clients to add content to the datastore as well as for rotating the secret aspects of the design via the HSM as required. Due to the nature of the environment Percival is intended to operate within, it is designed to operate while compromised. For example, it assumes that at least one query server has been compromised at all times, minimizes the release of information in the event that a client is compromised, and assumes that information is potentially leaked via communication channels despite the security of SSL, e.g. message contents remain hidden to an attacker but the size of a message is able to be detected.

In order to test its performance, we implemented Percival and ingested the Digital Corpora [22] consisting of approximately one million files of varying types, e.g. text, PDF, HTML. Search terms for each file were identified by performing a term-frequency inverse-document-frequency anal-
ysis of the word stems contained within each file. Percival’s search speed is based upon the time complexity of each query server’s key-value store and found to be less than one second. The cost of this performance is the space required at each site to store the secret-split reverse indexes, which is typically on the order of gigabytes.

1.3 Disaster Recovery

Despite the many challenges to maintaining data integrity over long periods of time [5, 72, 73], e.g. bit rot, technology obsolescence, and even natural disasters, secret-split datastores have the additional challenge of correctly reassembling the original data from the proper shares.

Ironically, the root problem stems from the main benefit of a typical secret-split datastore: it is information-theoretically secure [78]. As such, there is no inherent correlation between shares, and for this reason, systems such as this typically implement one or more methods to aid in share correlation during reconstruction, e.g. user maintained indexes [88]. Such indexes entail keeping a master list specifying which shares need to be recombined with which other shares in order to recover the original data. However, since the user maintained indexes method introduces a single point of both failure and compromise it is not recommended.

If normal reconstruction methodologies fail or are compromised, it is likely that data loss will occur due to the quantity of secret-split data requiring reconstruction. This is due to the exponential size of the reconstruction space needed to be processed during reassembly, which historically entails test reconstructing each piece of secret-split data, or share, from each server with every share from every other server. As a result, the process of identifying the correct shares with which to reconstruct each original piece of data is an exponential time algorithm.

The most recent disaster recovery method when these reconstruction aids fail is approximate pointers [88], which consists of each share keeping an ‘approximate’ reference to its next sibling share: shares split from the same piece of original data are called sibling shares. Unfortunately, even though it can identify ‘rings’ of sibling shares in near-linear time, its subsequent reconstruction phase has a high run time in order to determine the proper position of each sibling share. Furthermore, despite its high resistance to strict targeted theft, it has no resistance to loose targeted theft.

To address this need we present two novel disaster recovery methods: the Set-Subset method and the Secret-Split Hash method. These methods address the issue of data loss due to combinatorially prohibitive reconstruction. Both methods quickly reduce the reconstruction space needed to identify sets of sibling shares across servers without enabling strict targeted theft and with tunable resistances to loose targeted theft.
The Set-Subset method consists of selecting a random set of values for each group of sibling shares, and from that set of values, selecting a random subset of values for each sibling share within the group. Since the number of values selected for each share is known, potential false siblings can be filtered out prior to reconstruction by combining the sets from potential sibling shares and testing the size of that set. If this union of sets is larger than the known number of overall values selected for the group of sibling shares, these shares cannot be siblings.

The Secret-Split Hash method extracts a piece, or hint, of each share’s hashed ID, secret-splits the hint into hint shares, and stores a single hint share with each share of a group of sibling shares. Hint shares can then be recombined during reconstruction to reassemble the original hint and tested for validity against the share’s hashed ID. The first premise being that it is much faster to recombine many small, e.g. 8 bit, pieces instead of the relatively large shares. The second premise is that hints reassembled from different combinations of sibling hint shares must also match; this provides another mechanism for quickly reducing the reconstruction space.

Each method was first mathematically modeled to determine its theoretical performance. The models were then used to validate each method’s full implementation. Once validated, each implementation was analyzed for its runtime efficiency based on the number of shares per server, i.e. \( n \). The Set-Subset method was found to perform between \( O(n) \) and \( O(n^2) \), depending on the design characteristics chosen, whereas the Secret-Split Hash method was able to achieve linear runtime at the cost of decreased loose targeted theft resistance. In comparison to these new methods, the brute force reconstruction has a runtime of \( O(n^T) \), where \( T \) is the minimum threshold number of shares required for reconstruction.
Chapter 2

Background

Multiple aspects of Percival’s design utilize a non-key based form of encryption called secret splitting. As a result, it is critical that the reader have a basic understanding of how this unique form of encryption is performed. Furthermore, in an effort to minimize its space overhead, Percival does not support all forms of access control, but rather meets the needs of the most common production environment by providing support for only a subset of access patterns. As a result, additional information comparing the relative access control systems is discussed.

2.1 Secret Splitting

Secret splitting, or secret sharing, is an example of a threshold scheme \((N : T)\); it involves the act of splitting a piece of data, \(D\), into \(N\) pieces, or shares, such that only \(T\) shares are required for reassembly, where \(1 < T \leq N\). Thus, for example, a \((10 : 6)\) splitting scheme generates 10 equally-sized shares, where any 6 shares will enable the reconstruction of the original data. All ten shares are sibling shares of one another: a share is a sibling of another share if they are both generated from the same piece of original data, \(D\). A critical property of secret splitting is that, with less than \(T\) shares, no data is revealed.

In traditional secret splitting, \(T - 1\) random coefficients are computed for a \(T - 1\) degree polynomial, \(f(x)\), in a Galois field \([54]\) with order 256 or 65,536, i.e. the number of elements in the field. The characteristic of the field is typically chosen to be 2; subsequently, the number of elements is chosen by raising the characteristic to either 8 or 16 respectively. The constant term in \(f(x)\) typically contains the secret to be split. To generate each share, \(f(x)\) is evaluated using a random value for \(x\) between 1 and the order of the field. The share is then comprised of both \(x\) and \(f(x)\). To reconstruct the secret, at least \(T\) shares must be present in order to interpolate the values.
in the Galois field so that the constant term, i.e. the secret, can be calculated.

There are several known techniques to accomplish splitting a secret, all of which have varying levels of security. Shamir first introduced the concept of secret sharing [77]; Shamir secret sharing provides provable information-theoretic security, the cost of which being that each share is the size of the original data. Shannon defined information-theoretic security, or a perfect cipher system, where Equation 2.1 holds [78]. Specifically, that an attacker gains no knowledge about the plain text, \( M \), by knowing the cipher text, \( C \).

\[
I(M;C) = 0 \tag{2.1}
\]

There are several variants that strive to reduce the overhead usually required to achieve information-theoretic security, one of which is AONT-RS [68]; it combines an all-or-nothing transform with Reed-Solomon [67] encoding, resulting in a secret-splitting algorithm that is more efficient with regards to storage space. Each share is roughly \( d/T \) bytes, for a total storage of \( d \times N/T \), where \( d \) is the size of the data in bytes. However, AONT-RS is only computationally secure, not information-theoretically secure, since with fewer than \( T \) shares an attacker could guess a fixed-size key, thereby making it possible to verify whether those shares correspond to the data of interest.

While both of these approaches provide different levels of security, our design is agnostic with respect to the choice of secret-splitting algorithm, treating the algorithm as a black box. It is worth mentioning, however, that if a Galois field is used to facilitate splitting the data, e.g. Shamir’s secret sharing, the field prime should not be relied upon to be kept secret as this has been shown to cause information release despite forcing an attacker to reconstruct the data without the field’s characteristic [21].

### 2.2 Access Control

In this context, access control is used synonymously with authorization, and is defined as a system granting or rejecting access to a piece of data for an already authenticated user; as such, authentication is outside the scope of this thesis. In general, there are three main types of access control models: discretionary, mandatory, and role-based.

Discretionary access control (DAC) is the most common, as it is the default access control model of all UNIX systems. Access to a piece of data is defined the data’s owner, who is typically the user who caused the data to be created in the system. The owner can also assign access rights to other groups as desired. Mandatory access control (MAC) is the least common of the three standard models. It grants access to piece of data if there exists a rule to allow a particular user access. MAC
can either be rule or lattice based, and is typically used for government or military applications.

Role-based access control (RAC) is almost as common as DAC; it grants access to a user based on their role in the system, not on permission levels set by the owner of the data. An example of a RAC system is Cassandra [7]: a role based trust management system based on Datalog [1]. Becker et al. found that in real world applications role hierarchy and role delegation “occur in practice in many subtle variants [1].” As a result, Cassandra is designed to be able to express these subtleties easily instead of ad hoc, which is common in many languages.
Chapter 3

Related Work

Since this work touches on so many areas across storage-related research, e.g. trace studies, archival storage, encrypted searching, as well as search space reduction techniques, we now present an overview of the related work in each of these areas.

3.1 Trace Studies

There are two factors that make this study stand apart from previous trace file studies. First, this study addresses long term storage, which has vastly different workloads [45] and design requirements than those attributed to enterprise systems. For example, enterprise systems are often much more performance oriented. Second, this study analyzes the same site using the same analysis techniques twenty years later. We are unaware of other cases of multiple studies being performed on a single site, particularly with a separation of nearly two decades.

3.1.1 Enterprise Studies

There have been many trace-based studies conducted on enterprise and academic file systems over the past twenty years [6,25,46,71,91], none of which performed evolutionary trend analyses. However, taken as a whole, these analyses can suggest long-term trends in enterprise storage usage. Over time, file systems have grown dramatically in size, primarily by storing more files and a relatively small number of large files—individual file sizes in the traces have not grown as much. These trace studies also investigate user behavior at a relatively fine grain, since they can track individual users’ read and write behaviors across all files in the file system, not just those deemed important enough to archive.
A recent study performed on enterprise systems is the work done by Leung, et al. [46]. Despite the different organization and workload (day-to-day enterprise file system versus archival scientific file system) many of their findings were similar. In particular, both systems were found to have a low read-write ratio as well as a tendency for files to be accessed very infrequently. These findings serve to highlight the fact that, although enterprise storage systems are neither intended nor designed to be archival in nature, they may gradually become an archive by accident [97].

In contrast, the study by Gibson [32] on long-term behavior in a Unix file system and the trace analysis done by Agrawal, et al. on workstation file systems at Microsoft [4] are most similar in scope to the analyses conducted in this study. As expected, Agrawal, et al. found that, over the course of the study, many factors increased, including file sizes, file counts, file density (the number of files per directory), and others. Their study also showed that overall file age was not increasing, which is of particular interest because, if file age is not increasing and the rates for reading and writing stay constant, then the rate of deletion must increase or more storage must be added. For the system at NCAR, it is clear that the designers chose to increase overall storage; however, it is unclear whether workstation users do the same thing.

### 3.1.2 Long-Term Storage Systems

There has been relatively little study of usage patterns in long-term archival storage systems, perhaps because of the difficulty in gathering long-term traces. Many of the studies on archival storage systems for scientific computing were performed over twenty years ago [40, 81, 82, 90]. These studies had findings largely similar to the original study of the NCAR archival storage system [52], detailing usage patterns and user behavior. However, the largest archive studied in these systems was the 1992 NCAR archive, at 25 TB—the size of a workgroup disk array today. Given the advances in computing and storage technology over the intervening twenty years, the quantitative findings from these early studies are no longer relevant.

More recently, there has been renewed interest in understanding usage behavior of archival storage systems. Adams, et al. [2] found that many modern archival storage users modified files in the archive, and that non-scientific archive usage was very bursty. These findings differ from the characteristics of scientific archival storage systems, as this paper demonstrates.

### 3.2 Long-Term Storage

As previously stated, long-term storage brings with it a special set of problems due to the time scale in which it is required to operate. For example, even the organization hosting the archive
can act as a single point of failure. The LOCKSS [49] digital preservation system offers a possible solution to this issue due to its peer-to-peer design. However, data integrity becomes a major concern when working with peer-to-peer storage networks. LOCKSS addresses this by using many independent web-based repositories, each storing a copy of the data. These repositories are then coordinated through a complex polling scheme that guarantees “that even a group of very powerful adversaries attacking over many years have only a small probability of causing irrecoverable damage before being detected [49].”

Archives are also constantly plagued by space considerations. Historically the issue is mitigated by using strict retention policies, an alternative to which is the Presidio [99] system. Presidio dynamically selects between several space efficient methods in order to eliminate redundancy among data stored within the archive. It also hides these details from the user using a virtualized content addressable store so that users can freely access their data independent of how the underlying data was actually stored. There are many other modern archival frameworks, including Intermemory [15], Pergamum [89], and FARSITE [3], as well as schemes to increase reliability [16, 94] such as Store, Forget and Check [76], and Disaster Recovery Codes [34]. Each of these approaches addresses a specific concern. For example, Pergamum focuses on developing a low-power, cost-effective archive; disaster recovery codes offer an alternative to mirroring to increase reliability. Since this work focuses on making a small number of servers, that are not co-geolocated, secure yet usable, secret splitting offers the best all around starting framework on which to build Percival.

Secret split storage was first developed practically in the PASIS project [98], which also contains a good overview of $p-m-n$ threshold encoding schemes suitable for use in secret split archives. In response to several calls for providing archival storage that can operate through system compromises and provide resilience to insider threat [5, 87], this approach was later adapted for archival storage by Storer et al. in the POTSHARDS system [88]. Percival’s share storage most closely resembles POTSHARDS, though Percival uses a different approach for determining whether shares from different repositories can be combined to rebuild a data object. POTSHARDS also uses an index, stored in POTSHARDS itself, to track the relationships between shares; however, this index cannot be searched without the client first retrieving it.

Distributed systems that are resilient to large scale correlated failures, such as Glacier [37], often rely on massive redundancy. Typically this would result in unacceptable storage overhead. Glacier uses erasure coding and garbage collection to minimize this cost. While Percival does require a storage overhead equal to the size of the data times the number of repositories, Glacier typically requires eleven-fold storage overhead. However, its resiliency beyond that normally found in other threshold schemes illustrates the opportunity we have to investigate the resiliency of Percival, e.g.
refreshing a group of sibling shares back up to full strength in the event one or more them are lost.

An alternative to secret split archives was developed by Zage et al. [101]. In this approach, an algebraic-based encoding solution, Matrix Block Chaining (MBC), is used to “maintain data security and protocol performance when encoding large files. The design of MBC allows for encoding multiple partitions of the original data in parallel as subsequent encoding operations are not dependent on the output of previous encoding steps” [101]. Their technique was developed specifically for cloud storage, however, and as such does not maintain data availability in a compromised environment.

Pamies-Juarez et al. also saw the benefit in a decentralized archive based on erasure coding [23]. Their work focused on developing a “decentralized erasure coding process that achieves the migration [from a non-erasure code based, decentralized archive] in a network-efficient manner [23].” They were able to reduce the traffic during the decentralized encoding process by up to 56%. Since it will be more common to migrate an existing archive to Percival, rather than create a brand new archive, migration techniques will be vitally important for any real world deployment of Percival. Their work not only illustrates the need for migration techniques in general, but the need for ones specifically tailored to Percival.

For completeness, it is worth mentioning non-archive secure storage framework options on which to potentially build Percival. There are too many options to present here, and for the purpose of this discussion, a few examples will illustrate why they are not well suited to archival storage. The first example is Plutus [43], a cryptographically based storage system designed to operate in environments where little or no trust of the file servers exists. Its main contribution is a highly scalable key management system under the framework of discretionary access control, i.e. file owners dictate a file’s permissible access. The second example is SUNDR [47]: Secure Untrusted Data Repository. Like Plutus, it is designed to operate on untrusted servers and utilizes cryptographic keys, but its main focus is intrusion detection, specifically any attempt at an unauthorized modification to data. It is accomplished by adding a level of transparency between clients in order to share their file modifications. The last example is SNAD [53]: Secure Network-Attached Disks, the focus of which is to completely hide data from unauthorized users via strong encryption. An entity gaining physical access to the hard drive is unable to reveal any information about the stored data. Furthermore, it also contains intrusion detection in the form of detecting attempts to forge data. Despite their contributions, they are not well suited for archival storage because of the complications arising from key management, specifically “system policies, user training, organizational and departmental interactions, and coordination between all of these elements [96].”
3.3 Searching Over Encrypted Data

As our society shifts to be more digitally based, the amount of information that is being stored for long periods of time increases. However, as the quantity of information to store outstrips the local storage typically available, more often this information is being stored remotely. When this information begins to contain personal or sensitive information, privacy becomes a primary concern. However, privacy and accessibility are typically inversely proportional. This need to maintain both privacy and accessibility has given rise to many methods whose goal is to do just that. Song et al. [84] have developed several methods to address this need. While their design does offer security in the short term, it is not well suited for archival storage since, once the encryption scheme is broken, not only is the search history revealed, but the underlying encrypted data is as well. It is assumed that all key based encryption schemes have the potential to be broken given enough time.

Another alternative for privately searching over encrypted data is by keeping a local index that maps keywords to document IDs [14]. This index is then masked using pseudo-random bits and stored on the remote server along with the user’s data. When the user wishes to conduct a search, they provide the requested indexes along with short seeds in order for the server to recover small sections of the masked index. While this approach has potential, it is not appropriate for a secret split storage system since it introduces a single point of attack, which negates the security inherently gained by secret splitting one’s data.

Chang et al. [14] developed an approach using bit masked dictionaries to enable searching of encrypted remote data without revealing information to the data’s custodian. The outcome is similar to using a Bloom filter based system where a single bit is used to represent a term stored in the filter. The main difference is that it does not address conjunctive or disjunctive searches, nor does it address mapping multiple terms to the same bit in the dictionary.

A more modern approach to searching over encrypted data is to use homomorphic encryption [31], which allows for operations to be conducted on cipher text without the need to first decrypt the data. More precisely, given the “encryptions $E(x_1), \ldots, E(x_t)$ of $x_1, \ldots, x_t$, one can efficiently compute a compact cipher text that encrypts $f(x_1, \ldots, x_t)$ for any efficiently computable function $f$. The problem was originally posed by Rivest et al. in 1978 [31, 70].” Its application to searching would allow a user to encrypt the search terms, pass them to the server to then apply a homomorphic function to the stored encrypted data, and then pass the result of that encrypted operation back to the user for post-processing. This functionality comes at a price, however. Homomorphic encryption suffers from slow performance, primarily during the key generation phase [74]. Scholl et al. developed an extension to the Gentry-Halevi key generation technique [31] that is approximately twice as fast as the original, but despite this improvement, the algorithm is too slow for this application.
since even a relatively small delay becomes prohibitive when performed millions of times per search.

Given the distributed nature of the chosen storage framework, public key encryption (PKE), or a variant thereof, is an intuitive way to approach searching data in a secret split archive. A PKE solution to encrypted searching was proposed by Boneh et al. [10]. It is a straightforward, elegant solution that allows a user to provide a data store with an encrypted version of one or more search terms, thereby allowing the server to perform a binary go-no go test on the user’s data. Curtmola et al. [20] developed a variant of PKE that closely resembles homomorphic encryption. Their searchable symmetric encryption scheme also incorporates the ability for entities other than the data owner to perform searches over the encrypted data. In essence, a “search operation for a keyword \( w \) can only be performed by users that possess a trapdoor for \( w \) and […] the trapdoor can only be generated with a secret key [20].” If nothing is known about a trapdoor, no information is leaked by the encrypted document lookup index. Unfortunately, PKE solutions are not well suited for archival storage due to their reliance on fixed key encryption.

By way of comparison, Octopus [93], does not rely on encryption. It is an anonymous way for P2P nodes to communicate via a distributed hash table that provides a mechanism for individual queries to be sent along “multiple anonymous paths, [while introducing] dummy queries to make it difficult for an adversary to learn the eventual target of a lookup.” [93] In contrast to using dummy queries as a means of obfuscation, Percival’s distributed nature makes it irrelevant for an attacker to determine the eventual target of a search since even if a shard of interest is identified on a single compromised repository, it does not aid the attacker in performing a targeted theft of its sibling shares from other uncompromised repositories.

Bellovin et al. [8] addressed not only a similar approach to the problem of searching encrypted data, but in the same threat space as Percival. Specifically, when two agencies need to share documents, but only if the documents are requested. The crux is that the document store is not browsable by the requesting agency, and the providing agency must not have access to the requesting agencies’ queries. Bellovin et al. overcame these limitations by using Bloom filters stored by a semi-trusted 3rd party. Percival improves upon their solution by removing the reliance on this 3rd party.

3.4 Search Space Reduction

The related work on search space reduction algorithms is quite extensive, as such several articles have been written that summarize the work to date [51, 80]. Ferrari et al. [28] adapted these techniques to apply an iterative approach to efficiently reduce the search space for human pose
estimation. In a recent work, Hemalatha et al. [39] also applied an iterative search space reduction approach to develop a ‘Multi-Level Search Space Reduction framework for large scale face image database’ [39]. Their work resulted in an average accuracy of over 93% while reducing the search time required for database image queries by 66% when compared to the standard approaches.
Chapter 4

Evolutionary Trends in a Supercomputing Tertiary Storage Environment

It has been shown that secret splitting improves reliability, reduces the risk of insider threat, and removes the issues surrounding key management, and yet to date there has been little to no adoption of secret-split datastores in production environments. It is well understood that a distributed datastore needs to accommodate the access patterns and meet the responsiveness demands of its users, scale over time, and provide long-term security guarantees without the need to re-encrypt the data, rotate keys, etc. In order to understand the missing element as to why datastores of this type have little proliferation we analyzed the trends in access patterns, storage capacity requirements, and latencies in a real world environment over 20 years. It was found that a system must be optimized for write-heavy workloads since writes became four-times more frequent over the 20 year period. Furthermore, the decrease in read-density was expected, but did not become insignificant. This calls out for the need for improved search techniques as the quantity being accessed drops while the quantity of data stored continues to increase.

4.1 Methodology

The National Center for Atmospheric Research maintains a large supercomputer center whose primary responsibility is supporting climate researchers. These researchers use the archival storage system at NCAR to preserve both gathered data and the output of climate models over
long periods of time, providing a historical record of the climate research over the lifetime of the center. This information is used for several purposes. Typically, data cannot be analyzed in real time; instead, it is stored in the archive for later analysis. In addition, older data is sometimes used for comparison with more recent climate models and, in some cases, verification of older model results against observed conditions.

4.1.1 Evolution of the NCAR Mass Storage System

The overall design of the archive has not changed over the past two decades: the system still has a disk cache in front of a large amount of tape storage, as suggested by the ‘storage pyramid’ shown in Figure 4.1. However, the dimensions of the system have grown dramatically, from 25 TB in 1992 to over 30 PB capacity in 2010.

NCAR’s mass storage system has always consisted of three main levels. There is a controlling server that acts as the gatekeeper to the mass storage network. Behind the gatekeeper, the first level of storage is the disk cache, whose capacity in 1992 was 100 GB. By comparison, today the disk cache is 1000 times larger and is comprised of 500–750 GB hard drives.

The next level of storage is the primary tape silo, which in today’s system is a StorageTek SL8500 Tape Silo. In 1992, the primary tape silo was a StorageTek Automated Cartridge Sys-
Table 4.1: Information of interest contained in a single trace record.

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>event completion time in hours, minutes, and seconds</td>
</tr>
<tr>
<td>log record type</td>
<td>code word for the event type (e.g. read, write, or create)</td>
</tr>
<tr>
<td>host sequence number</td>
<td>ID tag for the event; when combined with the filename signify a unique event</td>
</tr>
<tr>
<td>data transfer time</td>
<td>time in seconds to complete the transfer</td>
</tr>
<tr>
<td>transaction time</td>
<td>time in seconds from start of event to completion</td>
</tr>
<tr>
<td>file size</td>
<td>file size in bytes</td>
</tr>
<tr>
<td>storage level</td>
<td>1 = disk cache, 2 = primary tape, 3 = second copy tape</td>
</tr>
<tr>
<td>filename</td>
<td>absolute path and filename</td>
</tr>
</tbody>
</table>

tem 4400 with 6000 IBM 3480-style cartridges, each with a capacity of 200 MB [52].

The last level is the manual tape drives that act as overflow and temporary storage for the primary tape silo. This layer is currently made up of 70 StorageTek T10000B drives fronting over 30 PB of tape storage, whereas in 1992 its capacity was only 25 TB of shelved tape.

Beyond the expansion in capacity, the biggest significant change as a result of innovations in storage technology is that, in 1992 the maximum file size was limited by the capacity of the tape cartridges to 200 MB. While there may be a similar limit today, it is less important because the tape cartridges have a capacity of approximately 1 TB, resulting in a limit for file size that is larger than most climate models produce. The impact of this change is discussed later in the paper.

4.1.2 Archive Storage Traces

The supercomputing center at NCAR maintains detailed trace records for their mass storage system; they use the traces both to assist in planning for upgrades to the storage system, to record the health of the system, and to serve as proof that a requested transaction took place. The traces only contain references to user-initiated activities, such as reads, writes, and migration between levels. However, they did not contain all of the records relating to data migration to a new storage system or to reads performed to check data integrity of stored data. While such reads may represent a significant load on an archival storage system [2], we were unable to include them in our analysis because of the lack of complete trace data.

The traces we obtained from NCAR were in ASCII format, and were designed to be easy to
generate using standard logging software. Traces are maintained in ASCII for several reasons. First, ASCII is easily human-readable; this proved to be a boon for us because it allowed us to diagnose issues such as a format change that occurred during the tracing period. Second, ASCII traces require no trace-specific translation application to convert the trace logs into a usable format. While this approach may consume slightly more space for uncompressed traces, compression tools such as gzip are very fast, removing any additional storage overhead while preserving the advantages of ASCII traces. Table 4.1 shows the fields of interest in a single trace record.

Before analyzing the data sets, we first scrubbed the traces to remove any events not of interest, specifically any non-user based event. We then cleaned them up to address a naming convention change that occurred part way through the latter trace period. This operation was necessary to get an accurate measure of both unique events and files in the system. The information we obtained to deal with this change mid-trace was obtained from Gene Harano at NCAR; without his help, we might not have been able to run the analysis. This problem highlights an issue with monitoring and trace collection, particularly for long-term storage: the system must record not only activity but also changes to the log format itself. Failure to do so may render long-term traces far less useful.

Once we had the cleaned-up traces from the 2008–10 trace period, we converted the traces from the earlier 1990–92 trace period into the same format, allowing us to run identical analyses on both sets of traces. This approach removed the possibility of differences in the results being based on differing assumptions when processing the data, potentially yielding inaccurate evolutionary trend conclusions. Furthermore, by rerunning the original analyses, we were able to verify the accuracy of our new analysis algorithms and tools against the results in the original paper [52].

4.2 Results

At the time of the 2010 traces, the NCAR system contained approximately 69 million files. However, this study includes only those files that were actually accessed (read or written) by users during each trace period. In particular, read and write events involved in migrating data to newer hardware were not included. There are also atmospheric data files that are typically not analyzed by scientists until after three to five years, which means their access period would not fall within either trace period. The presence of these files does not invalidate the findings of this study however, since these types of files were also present in the original study. Therefore a comparison of the archive at these two periods of time is still valid.

An overview of the activity recorded in the traces is shown in Table 4.2. Note that percent-
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>800K (66%)</td>
<td>7,424K (34%)</td>
<td>411K (33%)</td>
<td>14,502K (66%)</td>
</tr>
<tr>
<td>Disk</td>
<td>488K (60%)</td>
<td>3,509K (22%)</td>
<td>324K (40%)</td>
<td>12,431K (78%)</td>
</tr>
<tr>
<td>Tape (Silo)</td>
<td>162K (66%)</td>
<td>3,913K (65%)</td>
<td>82K (33%)</td>
<td>2,070K (35%)</td>
</tr>
<tr>
<td>TB Transferred</td>
<td>21.8 (72%)</td>
<td>1,805K (39%)</td>
<td>8.0 (28%)</td>
<td>2,780K (61%)</td>
</tr>
<tr>
<td>Disk</td>
<td>1.60 (55%)</td>
<td>617K (24%)</td>
<td>1.30 (45%)</td>
<td>1,924K (61%)</td>
</tr>
<tr>
<td>Tape (Silo)</td>
<td>13.1 (66%)</td>
<td>1,187K (58%)</td>
<td>6.5 (34%)</td>
<td>856K (42%)</td>
</tr>
<tr>
<td>Avg File Size (MB)</td>
<td>61</td>
<td>730</td>
<td>44</td>
<td>575</td>
</tr>
<tr>
<td>Disk</td>
<td>7.6</td>
<td>528</td>
<td>8.9</td>
<td>464</td>
</tr>
<tr>
<td>Tape (Silo)</td>
<td>182</td>
<td>911</td>
<td>177</td>
<td>1240</td>
</tr>
<tr>
<td>Latency (sec)</td>
<td>130</td>
<td>379</td>
<td>92</td>
<td>182</td>
</tr>
<tr>
<td>Disk</td>
<td>30</td>
<td>8.8</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>Tape (Silo)</td>
<td>103</td>
<td>151</td>
<td>74</td>
<td>83</td>
</tr>
</tbody>
</table>

Table 4.2: Overall trace statistics, with activity normalized to an annual basis.

ages shown are contributing percentages to the total events of that type. For example, read events to disk constituted 60% of the total number of disk references each year from 1990 to 1992, but only 22% each year from 2008 to 2010. However, summary values have been normalized to annual values to account for different trace durations for the 1992 and 2010 data sets. This means, for example, that there were about 7.4 million reads per year in the 2010 trace, and 21.8 TB of data read per year in the 1992 trace.

4.2.1 Read Density

The read density of a system is defined as the ratio of read events to write events for the system. In 1992, as Table 4.2 shows, the NCAR archive was read-dominated, with twice as many reads as writes. By 2010, however, writes dominated, with two times as many writes as reads. This change has serious implications for archival system designers, who should consider optimizing the
This change in user behavior likely results from one of two influences. First, users may simply be storing more data that they care less about, bolstered by the long-term decrease in storage cost. Because storage is much less expensive now, users can afford to be less selective in the data they choose to store; the process of making the decision is more expensive than at least the initial storage cost. The second possible explanation is that the rate at which users are accessing the data is staying the same relative to computing power, but the rate at which users are storing data has increased greatly. As described later, this explanation is supported both by the drastic change in the read to write ratio as well as the fact that users only accessed 30% of the current archive over the three year period of this study.

This change has far-reaching implications for system designers, particularly those involved in domains in which funding for a file’s maintenance is generated on a per-access basis, including advertising-supported sites such as video-sharing and photo-sharing sites that derive revenue from advertising displayed alongside access media. As the read-to-write ratio declines, the read density of the archive declines, providing less income to maintain the archive. Compounding the problem, the data in the archive must be migrated to newer media and devices, both to deal with aging devices and to leverage improved device performance. In addition, the decreased number of file reads also means that the system (rather than users) must initiate more read accesses to maintain data integrity and avoid bit decay [33]—infrequently accessed data must be explicitly checked by the system rather than implicitly checked during user accesses.

4.2.2 Inter-Reference Intervals

While the archive at NCAR is increasing in size, many of the files in the archive remain unaccessed for long periods of time, as Figure 4.2 shows. We can measure the amount of archive activity as well as the amount of reuse using the inter-reference interval: the time between successive accesses to a particular file. Files that are only accessed once do not have an inter-reference interval; such files are excluded from this graph.

The read and write events for a given file were broken down by event type and sorted by their timestamp. The inter-reference interval was then determined for each event type. Figure 4.2 shows a comparison of the inter-reference intervals for each event type as well as the overall inter-reference interval for both the 1992 system as well as during the 2010 traces. The most drastic change between the 1992 archive and the current one is that in the past, 80% of the system was accessed more than once, whereas only 30% of the current system is being written to, and only 8% of it is ever being read. It also illustrates the dramatic shift in read density between the two trace
periods.

This shift over the past twenty years allows designers to implement policies for files that haven’t been accessed within a certain amount of time forcing them to be offloaded to a slower and less expensive storage medium since the probability of accessing that file in the future is almost zero. These findings are further supported by those found during the read density analysis in section 4.2.1.

4.2.3 Latency to First Byte

While most computer hardware has gotten orders of magnitude faster over the past twenty years, storage hardware has not seen similar performance increases. In particular, tape hardware—both drives and robots—have much higher bandwidth, but positioning delays (seek time, robotic load time) are not much lower twenty years later.

The average delay between when an event is requested and when the first byte of data is sent or received is the latency to first byte. With regard to the NCAR traces, the latency to first
Figure 4.3: Comparison of latencies to first byte. All latencies have improved with the exception of primary tape, whose latency worsened by a factor of 1.5.

byte was calculated by subtracting the data transfer time from the total transaction time. Figure 4.3 shows a comparison of these latencies between the various types of storage devices for both the 1992 system and the current one. As the figure shows, latency for most types of storage devices decreased between the two traces, though only by a small factor. This is not unexpected, since storage devices have not dramatically reduced positioning delays over the past two decades.

However, note that primary tape latencies are slower in 2010 than in 1992 by a factor of approximately 1.5. 85% of tape silo requests completed within 3 minutes in 1992, as compared to only 70% of similar requests in 2010. This disparity can be explained in several ways. First, the fact that tape requests are so small in the first place indicates that many requests to tape are for files on a currently-mounted tape; fetching a tape takes far longer than 3 minutes. As tapes become larger, however, even files on a single tape may be separated by a large seek, increasing latency. Second, the larger number of files in 2010 make it less likely that the file being requested next is actually sequential on tape, again necessitating a (slow) tape seek.

It is safe to assume that overall system performance has not decreased over time due to this increased latency; otherwise, users of the system would likely demand that this problem be addressed. One explanation for this increasing latency being masked is the increased performance of the disk cache, which is much larger and faster than it was in the original system. As a result,
the system still appears responsive to the typical user, even though accesses to the primary tape silo have actually gotten slower. This allowed NCAR to focus their monetary investment into ensuring that the disk cache meets the needs of their users while allowing the tape silo to be slower without impacting user access times.

This increase in tape silo latency has other implications as well. It is possible that performance could be increased by decreasing the seek times for tape. Both hard drives and tape silos have initial start up costs [102], but the seek times for tape drives are much longer.

4.2.4 Hourly Usage Patterns

Figure 4.4 shows that it may be better to optimize modern archival systems for a write-intensive workload rather than fully optimized for read. This is a large shift from the workload of twenty years ago that was primarily read-oriented. Write events can simply be cached during peak read times, and sent out to the archive during off hours. This approach allow the system to remain responsive to users during working hours, and then catch up on write activity when the read workload drops back to baseline.

In both the 1992 and 2010 traces, overall system load is highest during working hours: 8 AM to 5 PM, with the gradual decline in system load later in the day most likely due to a percentage of the staff working past normal working hours. This shift is primarily driven by read events and read bandwidth. Given the highly cyclical pattern of reads, and the relatively low level of read events and read bandwidth on nights and weekends, it seems clear that reads are primarily driven by interactive users rather than batch processing.

On the other hand, write workload stays high through the week in both data sets, even though the number of write events is much lower during the weekend. This implies that batch processing is a key driver behind large write events, as would be expected: supercomputing applications produce large files that are subsequently written to archive. However, in 2010, write event rates are somewhat decoupled from write bandwidths, indicating that different types writers may have different behaviors with respect to file size. More specifically, batch writes appear to put twice as much bandwidth load on the system as user-initiated ones.

Another major difference between the two trace periods is that, in the original system, read events were comparable to write events in the amount of data transferred, whereas in the current system write events make up a much larger percentage of the overall data transferred. This supports the conclusion that the system should be designed for the relatively constant write workload, with concessions taken to handle the spurious read traffic.

The approach of sizing for a constant write bandwidth and buffering writes during periods
(a) 1992 workload. Note that the write workload stays relatively constant throughout the week, even though the number of write events increases by 40% during the workday. Read workload follows the number of read events across the entire week.

Figure 4.4: Comparison of the average amount of data transferred per hour to the number of read and write events per hour. 0 is Sunday at midnight.

(b) 2010 workload. The write workload tracks the number of write events throughout the week, but not during the weekend, where the number of write events drops off, while the bytes transferred remain constant. Read workload follows the number of read events throughout the week, but the amount of data transferred per event during the workweek is more than double than during the weekend.

of high read is bolstered by the observation that, in 2010, batch reads typically put little load on the system, as shown by the low read rates on weekends. However, read rates spike during weekdays, likely due to users retrieving large archived data files for analysis. Thus, postponing writes during the day will not require excessive amounts of disk for buffering, making it more feasible to use this approach to reduce the required support level for concurrency in the system.

4.2.5 Weekly Usage Patterns

The amount of data transferred each week in the original archive and the current archive is shown in Figure 4.5. These graphs further support that conclusion that the system should be
designed for the write workload: in 2010, the amount of read data is well below write data, in contrast to the 1992 workload, where reads dominate writes.

During the two year period in the original archive, a ramp up of workload can be seen; this is expected, since the archive came online shortly before the trace period began. One thing to note, however, is that over the two year period the write rate does not increase with the read rate. One explanation for this was that the computing center was already running at full capacity, so reads were primarily due to user-oriented tasks such as visualization [52].

In contrast, in the current archive, there is no ramp up, but there is a steady linear increase in workload. Furthermore, the write rate is considerably higher than the read rate, whereas in the original archive the converse is true. Researchers are storing much more information than they are accessing to analyze, so the system must be designed to handle this type of workload.

Another observation is that the drop in system load over holiday periods is much more severe in the 1992 trace data as compared to the current one. This is most likely due to heavier use of batch processing, which would keep the system loaded without direct user interaction. Recall that Figure 4.4b illustrated the decreased contribution that read events have in in the overall system workload for the week.

4.2.6 File Sizes

When file sizes increase, keeping the rest of the system constant, it follows that overall performance will decrease due to a degradation in the overall parallelism of the system. Since the number of files accessed is not growing linearly with the average file size, it follows that the number of spun-up drives per event is also not increasing linearly. Therefore, since more data is being read or written per event per spun-up drive, simply due to the increase in file sizes the system will become more serialized. This increase in serialization will have a direct negative impact on system responsiveness.

Figure 4.6a shows a comparison between the sizes of files in the system in 1992 and 2010. One thing to note is that, beyond the expected result of file sizes getting larger and there being more files per directory [19], the reason the largest file size in the original system was 200 MB is due to the physical limit of the storage media at the time. The largest tape in the original archive was 200 MB. As a result, there are many files of up to 4 GB and larger in the 2010 system; 5% of files are larger than 1.3 GB.

The difference is even more dramatic when we consider the amount of storage used by different-sized files, as shown in Figure 4.6b. This distribution indicates that it might be useful to permanently archive small files on disk [89] rather than migrating them to tape.
Figure 4.5: Data transferred per week in the two archives. In 1992, overall workload increased over the trace period, but write rate did not. In 2010, there was a relatively constant workload across the trace period, including holidays, which implies an increase in the use of batch processing.
Figure 4.6: File size comparisons in the two archives. The maximum size in the 1992 archive was 200 MB due to the physical limit of the storage media at the time. Most of the space in 2010 is consumed by large files, even though most of the files are small.
4.2.7 Directory Density

Increases in directory density can cause performance issues due to the non-trivial task of searching a given directory. However, even though according to Figure 4.7 the number of files has roughly doubled between the 1992 and 2010 studies, the number of files per directory in the NCAR system does not approach the issues associated with immense data structures [58]. Whether the small number of files per directory is due to system limitations influencing users or an intrinsic property of the workload is a subject for future investigation.

4.3 Summary

By comparing trace data for the NCAR center from 1992 to trace data taken from the current archive we were able to determine evolutionary changes over the previous twenty years. The study produced several key findings that are relevant for designers building archival storage systems.

First, writes have become four times more frequent relative to reads over the past twenty years. This, combined with the reduction in the fraction of the archive that is actually accessed over three years, indicates that archives are becoming increasingly “write-only”, with attendant implications for system design.

Second, as expected, access latencies did not decline very fast. Given the bursty nature of
reads, it may be useful to design systems to group or pre-fetch data to reduce perceived latency, even if doing so means reading data from archive to disk cache that may never be used. A relatively large disk cache could also be used to hide this latency from users, perhaps even permanently caching small files to reduce access latency for them.

By studying the same storage system being used for the same purpose at two different periods separated by nearly two decades, we have provided valuable insight into long-term archival storage system behavior. Furthermore, these findings guided the design of Percival, which is discussed in the following chapter.
Chapter 5

Percival: A Searchable Secret-Split Datastore

Building upon the findings presented in the previous chapter, we designed and built Percival, whose purpose is to provide secure searching capability for a distributed secret-split datastore in an untrusted environment. The main concern of the system is to maintain information privacy one trusts the whole, but not the individual storing one’s data. A secret-split datastore is inherently unable to be searched without first reassembling the data contained therein. It is for this reason the current solutions pre-index the data, and then implement search upon these indexes. However, these solutions all rely upon fixed-key encryption, which has been shown to be not well suited for long-term storage. By pre-generating search results, and then storing those results secret-split and distributed to a meta-layer of a distributed storage system, Percival is able to deployed to an existing system with little or no disruption to its operation. Furthermore, Percival is able to perform searches upon this information-theoretically secure layer while meeting the access patterns and responsiveness requirements of its users.

5.1 Design Overview

It is important to have a clear, high-level, understanding of Percival so that details of its design are framed in the proper context. For this reason, we now present a general overview of the architecture and information flow within Percival.
5.1.1 Query Server and Datastore Overview

Percival consists of two main layers: a query layer and a data layer. The query layer represents the collection of query servers: the secure key-value stores responsible for handling search requests from clients. The data layer is not restricted to being a secret-split datastore, e.g. POTSHARDS or Cleversafe. However, since it is reasonable to assume that an application that warrants the use of Percival would also warrant a secret-split datastore, we assume that the underlying datastore is also secret-split. Furthermore, Percival does not depend on the exact implementation of the datastore, it can be applied to both new and existing applications. Figures 5.1a and 5.1b illustrate two such possible distributed configurations. For brevity, we assume a POTSHARDS type configuration where there exists a collection of data servers to which the client saves and requests data objects. However, the reader should feel free to replace the data layer with any desired secret-split datastore; it will have no impact on the overall design or security of the system.

5.1.2 Information Flow Overview

During normal operation, clients are able to talk to both query and data servers, but the only in-band communication between any of the servers occurs during disaster recovery. Figure 5.2 shows the flow of information in the system during a search. In general, a client begins the search process by sending its search request to each query server, who responds with the correct share of a secret-split reverse index, or a non-empty response signifying to the client that there is no data that satisfies the search request. Reconstructing the reverse index reveals the identifiers for data objects that satisfy the search. After which, the client can either make additional queries, or, in the case where it has identified the data of interest, request the data object’s shares from the data servers using the appropriate identifier(s).

5.1.3 Splitting Scheme Flexibility

Two important design considerations when performing secret splitting are how many shares to generate and setting the reconstruction threshold. Recall that the reconstruction threshold determines the minimum number of shares required in order to reconstruct the original data. The number of shares to generate can be influenced by factors such as performance requirements, i.e. more shares results in more processing time, or the number of available sites at which to store the shares. The threshold is typically set based on the tradeoff between security and availability. A high threshold means more sites need to be compromised before data is leaked to an attacker, but if set too high, it increases the risk of a denial of service attack. For example, if the threshold is set to
require 9 out of 10 shares to reconstruct the data, it becomes more difficult for an attacker to obtain
the required number of shares. However, if just two sites are compromised, an attacker can corrupt
or otherwise modify those shares such that it is impossible for anyone to obtain the required number
of valid shares, thereby successfully executing a denial of service attack and ultimately data loss.
An overly high threshold also increases the risk of a ransomware attack [100]: the attacker encrypts
the data, and then ransoms it back for a price.

In contrast to a high threshold, a low threshold improves availability because a denial of
service attack is much more difficult to carry out, but the low threshold increases the risk of leaking
Figure 5.2: Overview of information flow during a query. (1) A client starts the process by sending a search request to each of the query servers. (2) Each query server responds with the correct share of a secret-split reverse index, or a non-empty response signifying ‘not found’. (3) The client can then either make additional queries, or proceed and request the data of interest from the data servers by sending the appropriate identifier(s). (4) Each data server then responds with its share of the secret-split data object.

Information due to the few number of sites that need to be compromised in order to reach this lower threshold.

Percival provides a level of flexibility with regard to these design considerations because it separates the reverse indexes from the stored data objects, i.e. the query layer from the data layer. As a result, different secret-splitting schemes can be used to address different operational needs or security requirements for each layer.

5.1.4 Access Control

Authentication is the linchpin of any security system, the compromise of which results in unauthorized data release and potential data loss. External to Percival, we assume there is a secure framework to which clients authenticate. Percival then uses the provided credentials to segregate the system such that the data loss due to a compromise of said credentials is as localized as possible. Specifically, Percival accomplishes this by creating a unique set of reverse indexes for each authentication credential. Note that access control does not apply to Percival’s claim regarding fixed-key encryption. A compromised access control credential can potentially result in a release of information in both Percival and standard fixed-key encryption based systems.
If there are a large number of such credentials, this can potentially result in very large space overhead. To address this issue, we assume that role based access control (RBAC) is used as the authentication framework, since it has been shown that on average, organizations have roughly 20 roles defined in their RBAC system [27]. This minimizes the space overhead while allowing the system to be segregated to localize the potential data loss.

5.2 Query Server

We now present the design and operational details of a query server, including its role during ingestion, specifics about its hardware security module, and how it performs salted hashing to facilitate search operations. Recall that we define a query server as a hardware security module backed by one or more machines working together as a single, logical key-value store, all contained within a single geographical location. Our use of a key-value store is for clarity and brevity. It is possible, and likely, that a real-world Percival installation would use a suitable replacement. For example, the secure partitioning scheme as defined by Parker-Wood et al. [57] can be used as a secure and efficient way to store and quickly access the query server’s shares.

5.2.1 Ingestion

The first step during ingestion is to identify the keywords for each data object. For our purposes, keywords are synonymous with search terms. The set of keywords for a data object define how that data object can be retrieved as a result of a search operation, i.e. how clients will search for their data using search terms. Once the set of keywords has been identified for each data object to be ingested, the reverse indexes for the corpus are generated by turning the mapping of data object to keywords to become a mapping of keyword to data object(s). Each of these reverse indexes can be viewed as a successful result to a search operation. This operation is illustrated in Figure 5.3.

The next step during ingestion is to secret split each of these reverse indexes, and then distribute the resulting query shares to the query servers such that no two sibling shares are stored on the same query server. Recall that sibling shares are shares that originated from the same piece of data, or reverse index in this context. Figure 5.4 is an example of sending a single query share and its keyword to be stored on query server \( n \).

The (keyword : RBAC) credential pair is sent directly to the HSM over its own network interface, while the share is sent directly to the key-value store backend of the query server via its separate network interface. This is to ensure that the actual keyword is never exposed to the query server, and is instead pre-processed by the HSM. During this pre-processing, the HSM performs a
Figure 5.3: Generating reverse indexes. Once the keywords for each data object have been identified, they are transformed into reverse indexes mapping keywords to their data object(s).

Figure 5.4: Overview of ingesting a query share. The keyword and RBAC credential are sent to the HSM, while the query share is sent directly to the key-value store. Once the keyword has been processed into a hash string, the resulting hash and query share are stored in the server’s key-value store.

The hash of the keyword and the provided RBAC credential, which is the same credential required by an authorized client in order to retrieve this query share in the future. The HSM then performs one more salted hash using the query server’s unique salt. Section 5.2.3 discusses the salt in detail. For now, it is sufficient to understand that each salt is kept secret by being generated by, and never leaving, the HSM, and its purpose is to ensure that each query server’s contents are unique. Once
the HSM has completed its pre-processing, i.e. the keyword has been transformed into the final hash string, the resulting hash and query share are stored in the query server’s key-value store. The HSM also stores the keyword internally to be used when performing salt rotation.

5.2.2 Hardware Security Module

Each HSM is a commercially available, physical piece of rack-mounted hardware that has its own network interface. Typical applications for an HSM include [55]:

1. The key generator and safe key storage facility for a certificate authority,
2. An accelerator for SSL connections,
3. A tool for securely encrypting sensitive data for storage in a relatively non-secure location,
4. A secure key generator for smartcard production.

Percival leverages the HSM to handle all server-side hashing as well as safeguarding the query server’s salt. The HSM provides both tamper evidence as well as tamper resistance in the form of logging and clearing its internal memory if it detects a potential intrusion. The price, however, of this secure crypto-processing is limited internal memory and low bandwidth. Typical models have an internal memory of roughly 10 MB and a bandwidth of less than 1 MB/s [48]. However, since Percival separates search processing from the actual data objects, this bandwidth limitation only impacts the rate of incoming search requests from clients, which is easily overcome by placing additional HSMs in parallel on each query server. As a result, Percival easily scales based on the size of the expected client base. Table 5.1 lists two examples of commercially available hardware security modules.

It is important to not confuse an HSM with a trusted platform modules, or TPM. A TPM handles similar job types, but is typically mounted directly on the motherboard, and as such, does not have its own network interface. As a result, all communications to and from the TPM must go through the motherboard, which would break Percival’s security model since Percival relies on the HSM to keep its processing completely isolated from the rest of the query server.

These modules also feature a means of secure communication between HSMs that provide the same security guarantees as the module itself, which allows multiple HSMs to share the contents of their internal memory without compromising the security guarantees of the HSM. This feature is leveraged by Percival in two ways. First, during initialization and salt rotation, it enables a single HSM to generate the query server’s salt and share it with the other HSMs. Second, the HSMs maintain a list of all keywords in the system in order to facilitate hash updates during salt rotation.
Table 5.1: Examples of commercially available hardware security modules [48].

<table>
<thead>
<tr>
<th></th>
<th>Infineon SLE 88</th>
<th>IBM 4764</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>66 MHz</td>
<td>266 MHz</td>
</tr>
<tr>
<td>Memory</td>
<td>16 KB</td>
<td>32 MB</td>
</tr>
<tr>
<td>I/O</td>
<td>12 KB/s</td>
<td>9.85 MB/s</td>
</tr>
<tr>
<td>SHA-1 1KB</td>
<td>155 KB/s</td>
<td>1.42 MB/s</td>
</tr>
</tbody>
</table>

without having to store the keywords externally; this is desirable since storing this list externally is both an unnecessary administrative overhead as well as being a potential vector for information leakage. The space requirement for this list is not large: on the order of megabytes, but it is also non-trivial given the limited available internal memory provided by typical HSMs. By placing multiple HSMs in parallel, less expensive HSMs with less internal memory can be used due to this secure communication channel. For simplicity, the rest of the paper assumes a single HSM is present on each query server, but the reader should keep in mind that they are easily parallelizable.

5.2.3 Secret Salt

Each query server has a single, unique salt that it keeps private. The salt is randomly generated by its HSM, and during normal operations, never leaves the HSM. However, since nothing is impossible to break, Percival takes this potential vulnerability into account and minimizes the information released in the event of a salt being exposed. Specifically, without additional elements being compromised, no information is released simply by determining the salt for a single query server. Section 5.3.2.1 discusses this threat model in more detail, as well as what other information is required prior to the privacy of the system being compromised.

The purpose of the salt is to ensure that the contents of the query server are unique: the corresponding hashes mapped to sibling shares stored on separate query servers are never the same. This is required since if the hashes were the same, an attacker would be able to identify sibling shares across query servers using only this hash string, and thus enable targeted theft: the ability to steal small blocks of data identified by some characteristic, which is the identifying hash string in this case.

The salt is the basis for our claim that Percival is more secure than standard fixed-key encryption. Typically in such systems, if the key used to encrypt the data is compromised, it will result in a catastrophic release of information. Whereas in Percival, a compromised salt alone results
in no loss of information or privacy, and does not impact the security of the data or search privacy on the other query servers.

While it is theoretically possible to brute force a query server’s salt by performing an exhaustive key search, it is infeasible due to the energy required based on the Landauer limit [44], not to mention the time required to do so. The Landauer limit, $L$: the theoretical minimum energy required to erase one bit of information, is defined in Equation (5.1), where $k$ is the Boltzmann constant ($\sim 1.38 \times 10^{-23}$ J/K), and $T$ is the operating temperature in kelvins.

$$L = kT \ln2$$  \hspace{1cm} (5.1)

For example, in order to simply flip through the bits of a 256 bit salt would require $2^{255}$ bit flips. Assuming this occurred at room temperature, the Landauer limit states this would require at least $8.9 \times 10^{39}$ TWh, which is many orders of magnitude greater than all the energy that has ever been produced on the planet [11]. As a result, it is much more feasible to attack the salt via side channels as opposed to attempting to obtain it by brute force.

### 5.2.4 Salt Rotation

In the case that it becomes necessary to change a query server’s salt, Percival enables salt rotation such that it only impacts that particular query server for a short period of time. When initiating a salt update operation, the HSM is provided with a list of all available RBAC credentials. This allows the HSM to generate a new salt, and then simply iterate over both the provided list of credentials and the internal list of keywords present in the corpus, recreating the original hash as well as the new hash for each entry in the key-value store. These pairs of hash strings are then passed to the server’s key-value store so that it may update its hash table, replacing the old hash with the new one.

### 5.2.5 Performing A Query

To conduct a query, or search request, the client first hashes the search term with the user’s RBAC credential. This hash string is then sent to the query servers, via their HSM, which hashes the user’s input with the query server’s salt. The final hash string is then passed to the key-value store, who finds the query share, if it exists, and sends it back to the client. If no entry is found for the given hash string, the query server responds with a non-empty response signifying to the client that there is no data that satisfies the search request. The size of this ‘not found’ response is the same size as a reverse index share; this makes all responses, regardless of search outcome, the same.
size, thus preventing an attacker from differentiating between response types in the encrypted SSL stream.

It is worth noting that while Percival does not explicitly block use of the root RBAC credential, it is not recommended. Since root is, in essence, simply another RBAC credential, it is possible to ingest files and shares using root. However, this will lead to unexpected behavior when performing a query because using the root RBAC credential only grants access to those files that were specifically ingested using that credential, and no other files or shares. It is for that reason that while using the root RBAC credential will not negatively impact the system, its use is discouraged.

Once the client has received the set of sibling query shares, it is able to reconstruct the reverse index in order to obtain the list of data object identifiers, which are then used in accordance with the chosen datastore implementation, e.g. POTSHARDS or Cleversafe. Figure 5.5 illustrates this process.

Figure 5.5: Simple single search term query. A client starts the process by hashing the search term with its RBAC credential. The resulting hash string is sent to each query server’s HSM to be further hashed with that query server’s salt, and ultimately used to look up the requested share of the secret-split reverse index. Once the client has received the query shares, it is able to reconstruct the original reverse index: mapping the search term to the appropriate data object identifier(s).

Conjunctive queries are performed in a similar manner, beginning with a client sending a separate search request to the query servers for each search term. The client then reconstructs each of the reverse indexes obtained from the query servers and takes the intersection of the sets of data object identifiers. The resulting set of identifiers represents data objects that relate to all of the
5.2.6 Adding New Content

The process to add content to Percival leverages both the query and ingestion processes. After having identified the keyword(s) for the data object to be ingested, the client performs a query for each of these keywords, reconstructs the corresponding reverse indexes, and adds the data object’s identifier to each reverse index. Once the reverse indexes have been updated, they are secret split and sent to the query servers along with the hash of the related keyword and the user’s RBAC credential. This hash is sent to each query server’s HSM to be subsequently hashed with that query server’s salt, the outcome of which is stored in the key-value store along with the newly updated secret-split reverse index.

5.2.7 Concurrency

Percival uses strong strict two-phase locking [12], SS2PL, managed by a distributed lock manager, DLM, to resolve conflicts due to potential concurrent operations. Percival does not rely upon the particular implementation of the DLM, e.g. Chubby [13], only that it correctly follows the SS2PL protocol. The three operations in Percival that rely upon the DLM are salt rotation, query execution, and content addition.

During salt rotation, a write lock is obtained from the DLM for all entries in the query server’s key-value store. This prevents an add new content operation from causing an inconsistent state due to an update collision, and prevents a query operation from potentially obtaining an invalid search result.

When performing a query, if a query share has been locked for writing, the query server indicates this state to the client instead of responding with the query share. This can potentially cause a query operation to block if it doesn’t receive a number of query shares equal to or greater than the reconstruction threshold.

The last operation that can result in a potential conflict is when adding new content; during such operations, a write lock must be obtained for each entry to be updated. If this was not required, two conflicting, concurrent operations could partially overwrite each other such that a complete set of sibling shares for an entry no longer exists in the database. Specifically, the resulting set of shares for a single entry no longer contains true siblings, but instead is comprised of some shares from both operations.

The final outcome of the conflict depends on the nature of the race condition. Recall that in an \((N : T)\) threshold scheme, \(T\) shares are required for reconstruction. If the resulting set of shares
for that entry contained at least $T$ sibling shares, the end result would be diminished reliability and a loss of information from the other operation. Alternatively, if the resulting set of shares does not contain at least $T$ sibling shares, the end result would be the corruption of that reverse index due to its inability to be reconstructed; thereby obfuscating the presence of that search term in the datastore.

5.3 Threat Analysis

Percival’s main goal is to maintain information privacy in a distributed, untrusted environment. Furthermore, since its design is intended for long term storage, it is assumed that even though an attacker may have reasonable computing power and storage available, they have potentially unlimited time to carry out an attack. Each of Percival’s potential attack vectors are discussed in the following sections.

5.3.1 Client

The numerous side channel attacks in which a client may be compromised range from cold boot attacks [38] to social engineering [35] and everything in between. Client security is a large open problem that Percival is not trying to solve. Instead, we assume that one or more clients will be compromised and have designed Percival to minimize the damage to the system when that happens.

Compromising a client means that the attacker has access to the user’s RBAC credential, and as a result, is able to perform all actions to which that user has access, e.g. perform queries, add/modify reverse indexes, and ultimately access the data objects to which the user has access. They do not, however, gain any insight into how the query servers are storing their shares, have access to any salts, nor do they have any access to any information not related to that specific user’s role.

In order to recover from a compromised client, the user’s role credential must be changed. This can be accomplished by performing an operation similar to salt rotation, in which each HSM is given both the old and new credentials. It can then iterate across its internal list of keywords, generating old and new hash string pairs that the query server can use to update its key-value store.

5.3.2 Query Server

Compromising a query server can take several forms, including the system administrator who has an operational need to have access to an entire query server, or even the janitorial staff that needs physical access. It is common to have a single insider at a site, but not for multiple insiders
to work in close coordination [75]. That said, it is possible to have insiders unknowingly working in conjunction when guided, or manipulated, by a large external entity, e.g. a foreign government [18].

It is also more likely for an insider to control a single query server as opposed to multiple servers [88]. However, we relax this constraint somewhat by assuming that an attacker never controls more than $T - 1$ query servers, where $T$ is the threshold number of shares required for reconstruction. If $T$ or more query servers are compromised, there is nothing stopping an attacker from simply reconstructing all reverse indexes, thereby gaining access to the identifiers for all data objects in the datastore.

We assume that an attacker physically controls any compromised query server, and is able to run arbitrary code on them. In general, an attacker is able to read all incoming messages sent to the key-value store, but has no access to those sent to or from the HSM. We also do not consider denial of service attacks, since they can be mitigated at design time by setting the correct reconstruction threshold. See the previous Section 5.1.3 for a detailed discussion on the tradeoffs when setting the reconstruction threshold.

We assume that standard methods for tracking user access patterns and detecting errant behavior are in place on each query server. It is for that reason that we assume that it is not possible to steal large amounts of information from a query server without detection, but that it is possible to steal a small amount of data. It is for this reason that steps are taken to ensure that no identifying features of the secret-split reverse indexes exist, e.g. ensuring hash strings in the key-value stores differ across servers, all secret-split reverse indexes are of equal size, no correlation exists between the shares’ meta-data, etc.

5.3.2.1 Hardware Security Module (HSM)

The security guarantees made by hardware security module manufacturers provide a high barrier for attackers to overcome. HSMs are not, however, inviolate. As a result, we acknowledge the potential for an attacker to possibly read the contents of an HSM [9]. If successful, this type of attack would reveal both the static information contained within the HSM, i.e. the internal list of keywords for the corpus and the salt for that query server, as well as dynamic search request information, i.e. the search terms and user credentials. This would obviously lead to a loss of privacy since an attacker would be able to read the incoming search terms as well as the hash strings in the key-value store in plain text, i.e. as keywords. Despite this level of information exposure, it does not expose any information about the underlying data objects, or their potential correlation to keywords because the reverse indexes are unable to be reconstructed without obtaining the correct shares from the other query servers. This is the primary way that Percival’s design does not lead to
a catastrophic release of information upon compromise.

While the release of all keywords in the corpus is not desirable, it is limited in regards to the amount of information that is released. This can be quantified using the Shannon entropy: the average unpredictability in a random variable, which is equivalent to its information content [96]. Shannon entropy, $H$, can be calculated using Equation (5.2), where $p_i$ is the probability of character $i$ appearing in the character stream. As a concrete example, the book Moby Dick [50] contains approximately 200,000 words and has a Shannon entropy of 4.55 [26,86]. In contrast, the book only has approximately 6,800 unique stemmed words, which drops the Shannon entropy to 3.15. This drop in Shannon entropy indicates a significant loss of information, which illustrates that the real data is indeed greater than the sum of its parts.

$$H = - \sum_i p_i \log_2 p_i$$  \hspace{1cm} (5.2)

If it is suspected that a query server’s salt has been exposed, rotating the salt will not regain any loss of privacy since the attacker would be able to correlate between the old and new hash strings. To recover from this type of event, the query server would need to be rebuilt by first clearing the key-value store and having the HSM generate a new salt, after which the HSM iterates over its internal list of stored keywords, requesting the appropriate sibling share from each of the other query servers using a process similar to performing a query, i.e. send a salted hash of the keyword and the specified RBAC credential, $h_{acl}$, to the other query servers. Once the HSM has the set of sibling shares for a particular reverse index, it is able to generate a new sibling share without affecting the existing ones [77]. The final step is to generate the salted hash to be stored in this query server’s key-value store using the $h_{acl}$ and the new salt.

### 5.3.2.2 Key-Value Store

Compromising a query server’s key-value store is the most basic, and probable attack vector for an inside attacker since it requires no other part of the system to be compromised; it also leads to the least amount of information being leaked. Such an attacker would be able to identify ‘hot’ shares, i.e. those shares that are accessed at a higher frequency than others. Without the relevant sibling shares, however, this information is of little use. If an attacker was able to identify the hot shares on at least $T$ query servers, this information could be used to perform a targeted theft of those shares. However, this would violate our initial assumption that at most $T - 1$ servers are ever compromised.

Since an attacker is able to read all incoming messages to the key-value store, information about a user’s location and search pattern can also be exposed, unless techniques are taken to hide
their origin IP address [24].

### 5.3.3 Communication Channels

Percival assumes that secure communication channels, e.g. SSL, exist between clients and query servers. It is not that we assume SSL is inviolate, but rather that secure communication, and SSL attacks in particular, are part of a larger open problem that is outside the scope of the project.

As a result, attackers are unable to read the contents of the encrypted data stream, but we assume they are able to perform attacks based on message size. This takes the form of potentially revealing the quantity of search requests for a client as well as whether a search request was successful or not. Depending on where in the system the attacker is listening, the former can likely be mitigated by masking the client’s IP address and routing [24]. The latter is not a threat because the query server responds with a meaningful ‘not found’ message that is the same size as a reverse index share, thus making a negative response indistinguishable from a positive one.

### 5.4 Performance

In order to test Percival’s performance, we implemented and tested the system running on five query servers; each server was running 64 bit Linux on four cores using 24 GB of RAM. We ingested the Digital Corpora [22] consisting of approximately one million files of varying types, e.g. text, PDF, HTML. It is important to keep in mind that Percival’s performance does not depend on the size of the data objects ingested, since once ingested, Percival is only concerned with the reverse indexes, and does not interact with the data layer of the system. Keywords for each file were identified by performing a term-frequency inverse-document-frequency [66], TF-IDF, analysis of the word stems contained within each file. The exact method of identifying keywords is not critical since Percival’s design does not rely on it, only that they are identified. Once the reverse indexes for the corpus were generated using these keywords, the reverse indexes were secret split using a (5 : 3) splitting scheme, i.e. five shares were generated for each reverse index, any three of which were required for reconstruction. These shares were then ingested into a BerkeleyDB database [56] as the query server’s key-value store. It took approximately 20 hours to secret split the reverse indexes, and an additional 6 hours to ingest the shares into each query server’s key-value store. While these durations are both very corpus and platform specific, they provide the reader with a general sense of the one time, ingestion costs associated with getting Percival online, servicing a corpus of roughly one million pieces of data.

In order to analyze the effect varying the number of keywords per data object has upon
both the size and quantity of reverse indexes, we performed the previous steps choosing 10, 20, 30, 50 and 100 keywords per data object. Figure 5.6 depicts a cumulative density graph for each experiment. As expected, it can be seen that as the number of keywords per data object increases, the total number of keywords present in the corpus also increases. It is also worth noting that 2/3 of the reverse indexes were found to contain a single data object identifier, and almost all reverse indexes were found to contain fewer than ten identifiers.

While these results are very corpus specific, they illustrate the potential space overhead required to secret split the reverse indexes since all shares must be the same size, i.e. the majority of shares must be padded in order to be the size of the largest shares in the corpus. If they are not, then their size becomes a distinguishing characteristic, which would enable targeted theft. For example, if shares are not padded to be of equal size, and there exists only a few shares that are X bytes, then an attacker would be able to identify and steal those shares, simply based upon their size, from non-compromised key-value stores.

With regards to search performance, search requests were found to complete in less than one second; the only variance in performance being due to random network issues causing a small number of resend operations. We measure Percival’s search performance using the completion time since standard metrics, i.e. precision and recall, are dependent solely upon the number and accuracy of the keywords chosen during ingestion, which are at the implementer’s discretion. Percival’s search performance is based on both the number of keywords and the size of the corpus. The number of keywords can potentially impact the hash table look up time, i.e. from the query server’s key-value store, and the size of the corpus affects the reconstruction time since it impacts the size of the query shares.

The bandwidth required to perform a search request is quite low and fundamentally based on the size of the data object identifiers. It can be determined by multiplying the identifier size by the number of keywords per data object times the average number of search terms per search request. For example, assuming an identifier size of 32 bytes and 100 keywords chosen per data object, a search request containing three search terms would require roughly 10 KB, with the majority of that traffic bypassing the low bandwidth HSM.

Since a single HSM is known to be the performance bottleneck, we also tested how long it took to perform the critical, HSM intensive operations in Percival. To complete a salt rotation, it took a single Intel 4764 HSM roughly two minutes to update the entries for approximately three million reverse indexes using a 256 bit salt. By contrast, it took 53 minutes for the same HSM to perform a complete query server rebuild. Both of these benchmark tests were performed for a single credential, or role. Recall that Percival minimizes the impact of an access control credential being
Figure 5.6: CDF of Reverse Index Size. For each number of keywords selected per data object, 2/3 were found to contain a single data object identifier, and almost all reverse indexes were found to contain fewer than ten identifiers. Since all shares must be the same size, these would-be small shares need to be padded in order to make them the same size as the largest shares, otherwise it enables targeted theft of those shares.

compromised by creating an independent set of reverse indexes for each credential. Continuing the previous example, a single set of reverse index shares would require 9.6 GB, which is not trivial, but since it has been shown that most organizations have approximately 20 access control roles defined, the space requirement has a reasonable upper bound, and most would require less than a TB per query server for most applications.

5.5 Summary

In order to maintain information privacy when sharing data across a distributed, long-term datastore operating in an untrusted environment, we have presented Percival. A system that is designed to be applied to new or existing secret-split datastores, operates while compromised,
minimizes insider threat, localizes data release upon compromise of an access control credential, and still provides accurate and timely search results; this is all achieved with a space overhead of a few terabytes per query server.
Chapter 6

Efficient Disaster Recovery
Methods to Prevent Data Loss in a Secret-Split Datastore

The 20 year trace study revealed that not only are datastores becoming write-dominated, but also that quantity of data being stored continues to increase, which typically not a cause for alarm. For a secret-split datastore, however, this can be problematic, and ultimately lead to data loss due to computationally prohibitive reconstruction times during disaster recovery. The current disaster recovery method, Approximate Pointers, mitigates the danger of this type of data loss, but at the cost of little to no resistance to loose targeted theft. In an effort to keep reconstruction feasible for even very large datastores, i.e. petabytes and above, while maintaining the desired security guarantees, we developed two novel methods, the Set-Subset method and the Secret-Split Hash method. Both methods were shown to provide customizable resistance to loose targeted theft at the cost of reasonable space overhead.

6.1 Quantity-Born Data Loss

As previously discussed, the standard reconstruction methodology will eventually fail since it relies upon external knowledge to perform the reconstruction. When it does, information loss is almost certain due to the quantity of secret-split data requiring reconstruction. With no other information to aid during reconstruction, it can be seen that as either the secret-splitting threshold or the number of shares per server increases, the collection of these shares can become so large
that data recovery is combinatorially prohibitive; this is due to all possible combinations of shares from each server being required to be tested for reconstruction. The effect of this can be seen in Figure 6.1.

![Figure 6.1: Given a file size of 1 MB, a reconstruction threshold of three, $T = 3$, and a reconstruction rate of 20 GB/sec, reconstruction would take approximately 1.5 years to reconstruct 10,000 shares, as seen in the blow up detail of the lower left section of the graph.](image)

Assuming a reconstruction rate of 20 GB/sec and a reconstruction threshold of three, $T = 3$, it can be seen that the time required to reconstruct 1 MB files increases exponentially with the number of files, or shares, per server. Furthermore, even with a relatively small datastore, e.g. 10,000 shares per server, if no additional information is known about the relationships between shares, it would take approximately 1.5 years to recover all of the data.

### 6.1.1 Approximate Pointers

The most recent solution, approximate pointers, comes from the POTSHARDS [88] project and consists of tagging each share with an approximate reference to the next sibling share in the group. That sibling share, in turn, has an approximate reference to the next sibling share. This
continues for each sibling share in the set, with the last sibling share having a reference to the first sibling share. In this way a ring of shares is formed, with each pointer being an approximate reference to the next share in the ring. Once a ring of shares is identified, the relative order between shares is known, but the actual position of each share has yet to be determined. Furthermore, the exact position of each share must be correct during reconstruction in order to properly reassemble the original data. For example, if shares A, B, and C have been identified as belonging to a ring generated from a (5 : 3) splitting scheme, these three shares must be tested in positions 1-2-3, 2-3-4, 3-4-5, 4-5-1, and 5-1-2. As a result, it is this second phase of its algorithm that incurs the highest performance cost.

Approximate pointers resistance to attack, i.e. strict targeted theft, relies upon an attacker being forced to query for shares that don’t exist within a server. For example, if an approximate pointer from one share points to twenty other possible sibling shares, where only one of them actually exists, then an attacker could be identified when forced to query for the non-existent shares.

While this method has been shown to identify rings of sibling shares in near-linear time, and is an improvement upon previous methods, it too has room for improvement due to its high performance cost due to repeated test reconstructions. In addition, its reliance upon false queries precludes strict targeted theft, but enables loose targeted theft.

### 6.2 Our Proposed Solutions: An Overview

Table 6.1: Relative comparison of the key features between the most recent method, Approximate Pointers, and our proposed methods to prevent data loss in a secret-split datastore. Overall, both Set-Subset and Secret-Split Hash methods perform better than Approximate Pointers. These tests were run on a 4-core, 64 bit Linux machine with 24 GB of RAM.

<table>
<thead>
<tr>
<th></th>
<th>Approximate Pointers</th>
<th>Set-Subset</th>
<th>Secret-Split Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Availability</strong></td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td><strong>Space Overhead</strong></td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Targeted Theft</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resistance</td>
<td>Strict</td>
<td>High</td>
<td>Immune</td>
</tr>
<tr>
<td></td>
<td>Loose</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td><strong>Runtime</strong></td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>
We have developed two novel disaster recovery methods, Set-Subset and Secret-Split Hash, in order to address the potential combinatorially prohibitive task of reconstructing a large amount of secret-split data. Both methods aim to quickly identify false siblings in order to reduce the reconstruction space while precluding strict targeted theft. However, each provides differing overall performance and resistance to loose targeted theft, and as a result, have particular use cases to which they apply. Table 6.1 illustrates the relative comparison of the features at which each method excels or does poorly. It can be seen that both the Set-Subset and Secret-Split Hash methods outperform Approximate Pointers with regard to efficiency and runtime, as well as having a higher resistance to loose targeted theft.

Availability refers to the method’s ability to withstand missing pieces and/or servers within the datastore. Space overhead is how much additional space each method requires. The next feature, targeted theft resistance, describes the method’s ability to prevent shares from being identified or differentiated from each other as to allow for theft of a set of sibling shares. Recall that strict targeted theft refers to an attacker identifying and accessing a small number of shares without detection using only a server’s public interface. Building upon this attack vector, loose targeted theft refers to an attacker not only using a server’s public interface, but also having the ability to access a small number of shares directly without detection. Efficiency refers to how quickly the method reduces the search space at each iteration of its algorithm, and runtime is a general indicator of the processing time required for each method given a constant size input.

### 6.3 Set-Subset Method

The Set-Subset Method method consists of tagging each share with a set of values such that the union of these sets can potentially be used to identify sibling shares. In general, this method relies upon the following parameters:

- **$M$**: defines the range of allowable values for each set, $S$, as 1 to $M$.
- **$S_i$**: a random set of values that is generated for each data item, $i$, in the datastore.
- **$P_{ij}$**: a random, proper subset of $S_i$ that is:
  - stored with each share $j$, and
  - $|P| = \frac{|S|}{2} + 1$.
- **$R_s$**: the ratio of $|S|$ to $M$.

To implement this method, first the maximum allowed value, $M$, is chosen for the datastore.
From the range 1–M, a random set of values is selected for each set of sibling shares; this set of values is \( S_i \). Furthermore, each server uses the value of \( M \) to generate the keys for its pre-filtering hash map such that the keys are tuples of every combination of two values from the range 1 to \( M \), and the values are groups of shares whose \( P \) sets each have the particular pair of values present. The purpose of each server’s pre-filtering hash map is to quickly identify, and thereby test, only those shares that have a required pair of values present in their \( P \) set.

In this way, it is possible to greatly reduce the initial reconstruction search space by pre-filtering out those sets that do not have the required pair of values in common. For example, since it is known that any true siblings must have at least two values in common, then if during reconstruction a particular share’s \( P \) set contains the values 2, 3 and 5, then the algorithm is able to skip all share groups except for those found by the tuples: \( \langle 2, 3 \rangle, \langle 2, 5 \rangle, \text{ and } \langle 3, 5 \rangle. \)

The ratio between the maximum allowed value, \( M \), and number of values chosen, \(|S|\), defines both the efficiency and loose targeted theft resistance of this method; this ratio is denoted by \( R_{sm} \), and the effects of varying \( R_{sm} \) are discussed in detail in Section 6.3.2. It is sufficient at this time for the reader to understand that as \( R_{sm} \) is lowered, performance improves at the cost of reduced resistance to loose targeted theft.

Figure 6.2 illustrates an example of this process: given \( M = 8 \), the random set, \( S_0 \), is randomly generated for its set of sibling shares; from this set, \( N \) subsets are randomly generated such that \( P \subset S, |P| = \frac{|S|}{2} + 1 \), and \( 1 \leq j \leq N \) where \( N \) is the number of sibling shares in each set. It can be seen that the size of \( P \) is designed to be as small as possible while minimizing the number of sets that need to be combined in order to make a determination regarding potential siblings. Specifically, by setting \( |P| \) as a function of \( |S| \), i.e. \( |P| = \frac{|S|}{2} + 1 \), \( P \) is as small as it can be while allowing a determination to be made after only two sets are combined. If \( P \) was any smaller, then more sets would have to be combined before being able to make a determination since the size of the union of two disjoint \( P \) sets would still be less than \( |S| \).

These sets, \( P_{01}–P_{0N} \), are each stored with their respective share. Collectively, each of these \( P \) sets will be used to identify, albeit not uniquely, a set of sibling shares. Once each share’s \( P_{ij} \) set is determined, a reference to this share is also placed in server \( j \)'s pre-filtering hash map under the keys for each combination of two values from this set, \( P_{ij} \).

**Theorem 1** Given sets \( P_0, P_1, \text{ and } S \), where \( 0 < |P_0| \leq |S| \) \( \text{ and } 0 < |P_1| \leq |S| \). If \( |P_0 \cup P_1| > |S| \), then \( P_0 \not\subset S \) and/or \( P_1 \not\subset S \).

**Proof** Suppose that the size of the union of sets \( P_0 \) and \( P_1 \) is greater than the size of \( S \). This implies that one or both of the \( P \) sets contain(s) at least one element that is not present in set \( S \).
Case 1 \( P_0 \) contains at least one element not present in the set \( S \). Therefore by the definition of subset, \( P_0 \) cannot be a subset of \( S \).

Case 2 \( P_1 \) contains at least one element not present in the set \( S \). Therefore by the definition of subset, \( P_1 \) cannot be a subset of \( S \).

Case 3 Both \( P_0 \) and \( P_1 \) contain at least one element not present in the set \( S \). Therefore by the definition of subset, neither \( P_0 \) nor \( P_1 \) can be a subset of \( S \).

Therefore, since all possible cases hold, the premise holds. Q. E. D.

Within each of these groups, false siblings can be further identified by taking the union of each candidate sibling’s \( P_{ij} \) set. In accordance with Theorem 1, if the resulting size of this union set is greater than \(|S|\), the two shares cannot be siblings. By repeating this process for each sibling to be considered, the reconstruction space can be quickly reduced. Figure 6.3 illustrates an example of testing the first candidate share from server 1, \( P_{01} \), with the second share from server 2, \( P_{12} \). Since the \(|P_{01} \cup P_{12}| > |S|\), these two shares cannot possibly be siblings.

Figure 6.2: Given a maximum allowed value of seven, \( M = 8 \), the random set, \( S_0 \), is randomly generated for a single group of sibling shares. From this set, the sets \( P_{01} – P_{0N} \) are randomly generated and are stored with their respective share.

\[
S_0 = \{1, 3, 6, 7\} \\
P_{01} = \{3, 6, 7\}, \\
P_{02} = \{1, 6, 7\}, \\
\ldots \\
P_{0N} = \{1, 3, 6\}\]

Figure 6.3: Taking the union of two candidate share’s \( P \) sets when \(|S| = 4\). This is an example of two candidate shares that \textit{cannot} be siblings since here: \(|P_{01} \cup P_{12}| > |S|\).

By way of comparison, Figure 6.4 illustrates an example of testing the first candidate share from server 1, \( P_{01} \), with the third share from server 2, \( P_{22} \). In this case, since the \(|P_{01} \cup P_{22}| \leq |S|\),
Given: $P_{01} = \{1, 6, 7\}$ and $P_{22} = \{1, 3, 7\}$

$P_{01} \cup P_{22} = \{1, 3, 6, 7\}$

Since: $|P_{01} \cup P_{22}| \leq |S|$ ≠ siblings

Figure 6.4: Taking the union of two candidate share’s $P$ sets. This is an example of two candidate shares that might be siblings since $|P_{01} \cup P_{12}| \leq |S|$.

these two shares might be siblings. False positives, such as this, improve the datastore’s resistance to a loose targeted theft attack at the cost of decreased performance. Furthermore, the probability of a false positive can be adjusted by setting $R_{sm}$, which is discussed further in Section 6.3.2.

6.3.1 Modeling and Implementation

Prior to implementation, a model of each method was constructed in order to both analyze the theoretical reduction in reconstruction space as well as to validate its implementation.

The Set-Subset method was modeled by first identifying the factors that probabilistically determine how much the reconstruction space is reduced during each step of the algorithm. The first step of which is to only test those groups of shares on each server that have at least two values in common by leveraging each server’s pre-filtering hash map. Recall that each server maintains groups of shares based on tuples of common values present in each share’s $P$ set.

In general, this method can be viewed as a series of set union operations, where an incoming set, $P_1$, is combined with an existing set, $P_2$, such that the existing set $P_2'$ is updated to be the union of the two sets, i.e. $P_2' = P_1 \cup P_2$. While the number of values in the incoming set, $P_1$, is always constant, it is clear that the number of values in the existing set increases over the life of the algorithm.

In order for two shares to be siblings, they must have a minimum number of values in common between in their $P$ sets. This minimum number, $x$, is defined by Equation 6.1, and states that the minimum number of values required to be in common between the two sets is primarily determined by how many values can be added to $P_2$ while keeping $|P_2| \leq |S|$.

For example, given $|S| = 16$ and $|P| = 9$, if at a certain point during the execution of the algorithm an existing $P_2$ set has 13 values due to previous operations, then in accordance with Equation 6.1 these two sets need to have at least 6 values in common in order to be potential siblings.

$$x = |P_1| - (|S| - |P_2|)$$ (6.1)
Since the required number of values in common between any two $P$ sets in order for them to be siblings can be determined, it is possible to determine the probability of those two sets having $x$ values in common.

In order to ultimately determine that probability, we start with the probability that a single, specific value is present in a particular $P$ set. This probability, $\rho(sv)$, is given in Equation 6.2, and is defined as the probability that the value is present in the $S$ set multiplied by the probability that the value is present in the $P$ set based on the given $S$ set.

$$\rho(sv) = \left( \frac{|S|}{M} \right) \left( \frac{|P|}{|S|} \right) \quad (6.2)$$

Next the probability that two particular values are present in a particular $P$ set, $\rho(tv)$ can be found by multiplying the probability of a single value being present, $\rho(sv)$, and the probability of a single value being present in a set of size minus one. This is defined in Equation 6.3.

$$\rho(tv) = \left( \frac{|S|}{M} \right) \left( \frac{|P|}{|S|} \right) \left( \frac{|S| - 1}{M - 1} \right) \left( \frac{|P| - 1}{|S| - 1} \right) \quad (6.3)$$

Equation 6.3 can be generalized for $x$ values present in a particular $P$ set as defined in Equation 6.4

$$\rho(mv) = \prod_{a=0}^{x-1} \left( \frac{|S| - a}{M - a} \right) \prod_{b=0}^{x-1} \left( \frac{|P| - b}{|S| - b} \right) \quad (6.4)$$

Finally, the probability that two $P$ sets have $x$ values in common can be found by taking the probability of $x$ values being present in one set, $P_2$ and multiplying that by the number of ways those values can be present in the other set, $P_1$.

This probability, $\rho(x)$, is defined and simplified in Equation 6.5 and Equation 6.6 respectively.

$$\rho(x) = \left( \frac{|P_1|}{x} \right) \prod_{a=0}^{x-1} \left( \frac{|S| - a}{M - a} \right) \prod_{b=0}^{x-1} \left( \frac{|P_2| - b}{|S| - b} \right) \quad (6.5)$$

$$= \frac{|P_2|^{(t/P_2)} (1 - |P_2|)}{M_{x-1}} \quad (6.6)$$

Finally in order to determine the probability that the two sets have at least $y$ values in common, sum the required values for $y$ as in Equation 6.7.

$$\sum_{c=y}^{P} \rho(c) \quad (6.7)$$

\[ ^{3} \text{In general, } x_y = \frac{x!}{(x-y)!} \]
The model was then tested by varying both the range, \( M \), and the size of \( S \), thereby altering the probability in Equation 6.7, which directly affects the number of sibling tests needed to correctly identify all sets of siblings. In order to determine the benefit of pre-filtering, the model was first tested with pre-filtering disabled, the results of which were then compared to the reconstruction performance with pre-filtering enabled. Figure 6.5 illustrates this effect. It can be seen that when pre-filtering is disabled, performance drops by over 50%. Furthermore, when \( |S| = 4 \) and 8, insufficient information is stored with each share in order for the Set-Subset to correctly identify sibling shares when each server contains close to \( 10^6 \) shares and more than \( 10^8 \) shares respectively. This is due to not enough information being stored with each share when the \( R_{sm} \) is too low.

![Graph](image)

(a) No pre-filtering.  
(b) Pre-filtering on pairs of values.

Figure 6.5: Performance comparison with no pre-filtering during reconstruction and pre-filtering on pairs of values present in each share’s \( P \) set. It can be seen in 6.5a, when pre-filtering is disabled and \( |S| \leq 8 \), insufficient information is stored with each share to allow the Set-Subset method to correctly identify all groups of sibling shares.

Figure 6.5b shows the model for \( M = 256 \) and the resulting number of sibling tests that are required for varying share counts per server and varying sizes of \( S \) when pre-filtering on pairs of values. In this context, a sibling test refers to taking the union of each share’s \( P \) set and testing for compatibility, e.g. for compatible shares, \(|P_1 \cup P_2| \leq |S|\) whereas for incompatible shares \(|P_1 \cup P_2| > |S|\). The increase in required tests when \( |S| = 4 \) and there are more than \( 10^9 \) shares per server is due to decreased effect that pre-filtering has at that \( R_{sm} \). Ultimately, even with pre-filtering enabled, the Set-Subset method is not able to correctly identify all groups of sibling shares when the share count per server approaches \( 10^{12} \) since there is simply insufficient information stored with each share. This figure also shows that pre-filtering limits the maximum sibling tests required, regardless of the
Figure 6.6: The Set-Subset method implemented and validated against the theoretical model. The black solid line denotes the mean performance with the sample population fitting a $t$-distribution. Increasing the size of $S$ slightly decreases performance with the benefit of improving resistance to loose targeted theft. The increase in required tests when $|S| = 4$ and there are more than $10^9$ shares per server is due to decreased effect that pre-filtering has at that $R_{sm}$.

$R_{sm}$, e.g. when $|S| = 32$, since the required number of sibling tests is equal for both $|S| = 16$ and $|S| = 32$.

The Set-Subset method was then implemented and validated against the theoretical model, the results of which are illustrated as the black line in Figure 6.6. Each experiment was run until the sample set’s variance fit a student $t$-distribution [69]. In a $t$-distribution, the variance is equal to $d/(d-2)$, where $d$ is the degrees of freedom in the experiment, i.e. the number of test runs minus one. It can be seen that the theoretical model accurately predicted how the implementation would perform.

Prior to this point, all pre-filtering has been accomplished with each server’s pre-filtering hash map using a key size of two, i.e. store shares based on pairs of values present in their $P$ sets. In order to validate this approach, the model was then tested by increasing the key size to three, four,
and five; the effects of which are shown in Figure 6.7. It can be seen that increasing the key size beyond two has no impact on performance, which follows since the full effect of a pre-filter is only realized once the average number of values required to be in common between $P_1$ and $P_2$ is larger than the key size. For example, when pre-filtering the first server’s shares, two values are required to be in common between $P_1$ and $P_2$. If shares were instead pre-filtered using a key size of five at this point in the process, valid candidate shares would be erroneously excluded, thereby invalidating the process. This forces server’s early in the process to use a key size less than the target one, and as a result the effect of the larger key size isn’t realized until later in the process. Therefore, a larger key size does not aid in reducing the initial reconstruction search space, where it would be more effective, and instead only applies later when its effects have little or no impact.
6.3.2 Runtime Analysis

Recall that the core problem we address is the potential data loss due to a full reconstruction of all secret-split data requiring an infeasible amount of time. Therefore it is critical to understand how the design parameters for each method affect their runtime as well their resistance to loose targeted theft.

Figure 6.8: The Set-Subset method roughly performs between $O(n)$ and $O(n^2)$ depending on the chosen values of $|S|$ and $M$. As one moves left on the curve, performance increases at the cost of decreased resistance to loose targeted theft. The shaded areas denote regions where the percentage of homogeneity is potentially low enough to enable loose targeted theft depending requirements of the operating environment.

All runtime graphs show a quadratic and linear growth rate line for the number of operations, i.e. the number of sibling or reconstruction tests, required given an input size of $10^6$ shares per server. The linear growth rate line is based on a best case scenario in which a full reconstruction of all secret-split data can be performed in a single pass of a single server. The quadratic growth rate line is when $T = 2$ and is the equivalent to the brute-force reconstruction, in which one must
attempt to reconstruct all combinations of shares from each server. In general, brute-force recon-
struction requires polynomial runtime $O(n^T)$, where $T$ is the minimum number of shares required
for reconstruction. The quadratic growth rate line is a special case of this general process.

The solid curve in Figure 6.8 shows the number of sibling tests required for various $R_{sm}$
ratios. Recall that $|S|$ is the number of values chosen from which each share’s $P$ set values are drawn,
and $M$ defines the range of those values. Logically, as the $R_{sm}$ decreases, each $P$ set becomes more
unique, and in turn increases the efficiency of the algorithm, i.e. the reconstruction space for sets of
sibling shares is reduced more quickly.

The cost of this performance increase is the increased susceptibility to loose targeted theft,
denoted by moving to the left along the dashed line in Figure 6.8, because as each $P$ set becomes
more unique, eventually it becomes trivial to identify a share’s siblings using their $P$ sets. This effect
is visible in Figure 6.8 as the $R_{sm}$ approaches zero. The red and yellow regions on the graph are
included as a rough guideline of an acceptable level of loose target theft vulnerability, vary with the
actual acceptable level for a given operating environment, and illustrate the $R_{sm}$ ratios that result in
5% and 10% homogeneity of a server’s shares. As the $P$ sets on a server become more homogeneous,
i.e. more similar, loose targeted theft becomes more difficult to perform without detection since a
larger percentage of the shares would need be accessed without detection. This is why these regions
are only guidelines since the actual operating environment would dictate its own levels of detection.

In general it can be seen that the Set-Subset method performs between $O(n)$ and $O(n^2)$
depending on the chosen values of $|S|$ and $M$. Ultimately the design choice of what $R_{sm}$ to implement
is up to the system designer, and is based upon both the size of the datastore and the environment’s
acceptable level of susceptibility to loose targeted theft.

Algorithm 1 defines the process of how the Set-Subset method reduces the reconstruction
search space. Line 2 initializes the algorithm to use the 1$^{st}$ server’s $P$ sets as the $existingSets$.
Line 7 then tests each of these $P$ sets against each of the $P$ sets from the 2$^{nd}$ server by testing the
size of their union set. If the test passes, the new union set is stored in a list to be used as the $existingSets$ during the next iteration. The algorithm exits when the sets of sibling shares have all
been identified, or fails due to having visited all servers in the system without successfully identifying
all sets of sibling shares.

6.4 Secret-Split Hash Method

The Secret-Split Hash method is similar to the Set-Subset method in that its main goal is
to quickly reduce the reconstruction space required to identify sibling shares in order to perform a
Algorithm 1 SetSubset Algorithm

1: function SetSubset
2:   existingSets = setsByServer[0]
3:   step = 1
4:   repeat
5:     for each set in existingSets do
6:       incomingSets = setsByServer[step]
7:       for each incomingSet in incomingSets do
8:         union = set ∪ incomingSet
9:       if |union| ≤ |S| then
10:          outgoingSets.push(union)
11:     existingSets = outgoingSets
12:     outgoingSets = new set
13:     step = step + 1
14: until |existingSets| ≤ filesPerServer

full reconstruction of all secret-split data. This method’s efficiency relies on the following premises:

1. It is much less expensive, with regards to processing time, to reconstruct a very small amount
   of secret-split data as opposed to the potentially large share size, e.g. 8 bits vs. 1 MB or
   more.

2. Reconstructions must be consistent across multiple subsets of shares in a potential sibling
   group.

To implement this method, each share is first identified with a unique ID. From a hash of
this ID, a small subset of bits are chosen to be the hint for that share. In a manner similar to the
Set-Subset method, this hint will be used to do small, quick reconstruction tests in order to identify
those shares that can’t be siblings.

Figure 6.9: Hint generation example: the hint, $H_1$, is generated by passing the share’s ID through
a hash function, and then extracting a predefined range of bits from the resulting hash string.

Since the hash function is only used to take an arbitrary size ID string and map it to a
string of known, fixed length, its only requirements are that it is a true one-way function and its output is roughly uniform over the output range. The number of bits chosen, i.e. the size of the hint, affects how quickly the reconstruction space is reduced for each hint reconstruction test; this effect is discussed in Section 6.4.3.

Figure 6.10: Example of generating hint shares by secret splitting the hint $H_1$. Each hint share, $h_{s_{ij}}$, is the hint share generated from the hint $H_i$ to be stored with share $j$ within the group of sibling shares.

Figure 6.9 illustrates an example of the hint generation process. The hint, $H_1$, is generated by passing its share’s ID through a 256 bit hash function, and then extracting a predefined range of bits from the resulting hash string. Once each hint is generated, it is secret split to generate $N$ hint shares. Each hint share, $h_{s_{ij}}$ is the hint share generated from the hint $H_i$ to be stored with share $j$ within the group of sibling shares, as shown in Figure 6.10. Once the hint share $h_{s_{ii}}$ has been generated for share $i$, server $i$ stores the hint share in its pre-filtering hash table using $h_{s_{ii}}$ as the key. Each server’s hash table will allow the algorithm to pre-filter out those shares that could not possibly be siblings while not enabling strict targeted theft.

Figure 6.11: A set of $N$ sibling shares, each with the appropriate set of hint shares, $h_{s_{ij}}$, where $h_{s_{ij}}$ is the hint for share $i$ to be stored with share $j$.
Figure 6.11 depicts the $N$ sibling shares that originated from a single original piece of data, each stored with their appropriate hint shares, $h_{s_{ij}}$. We denote $T_h$ as the hint splitting threshold. While the splitting scheme used to generate the hint shares has the same $N$ value as that used when splitting the original data, that is it generates $N$ total hint shares from each hint, the hint splitting scheme doesn’t need to, and most likely won’t, use the same splitting threshold used to split the original shares, i.e. $T \neq T_h$. In this way, adjustment of the parameters in this method does not affect the overall security scheme of the datastore, nor does this method’s implementation require an existing secret-split datastore to be reassembled and re-split.

During reconstruction, the hint as well as the hint shares from each of the candidate shares are used to test whether shares can be siblings. If the hints don’t match, they cannot be siblings. However, if they do match, then the candidate shares might be siblings. The details of reconstruction are discussed in Section 6.4.2, but for now it is sufficient for the reader to be aware that multiple hint reconstructions occur when testing each group of shares as potential siblings. In this way the search space is reduced as quickly and efficiently as possible.

### 6.4.1 The Metadata Variant

Until this point, this work has stated that the set of hint shares attached to each share is simply that, a set of hint shares. However, if the method’s resistance to loose targeted theft can be relaxed, a more flexible implementation of this method is possible. In this variant, each $h_{s_{ij}}$ can instead be a hint tuple, $H_T$, containing metadata as well as the hint share itself, e.g. $H_{T_{ij}} = [splittingThreshold, startingBit, hintSize, h_{s_{ij}}]$. In this way, a global hint splitting scheme need not be implemented, and instead can be customized as needed on a per share basis with regards to the splitting threshold used to generate this hint share, as well as the starting bit position and the number of bits constituting the original hint. It is worth emphasizing, however, that while these parameters are independent and free to be customized, the more they are customized, the more vulnerable to loose targeted theft the datastore becomes.

For example, by setting 50% of the hint splitting thresholds to one value, and the other 50% to a different one, e.g. three and four respectively, an attacker now knows which half of the datastore that any potential sibling of a share of interest resides. This, of course, is a trivial example, but as more of the metadata is varied across sets of sibling shares, it makes each grouping of shares that have similar values more differentiable from the other shares. Furthermore, if these parameters vary often enough as to make the hint tuple unique, or simply unique enough, as to allow for trivial identification of a set of sibling shares, there would be no resistance to loose targeted theft.
6.4.2 Modeling and Implementation

While the Secret-Split Hash method has the same goal as the Set-Subset method, i.e. to reduce the reconstruction space required to determine sets of sibling shares, it operates very differently. Specifically, whereas the Set-Subset method’s efficiency is based on the probability of set commonality, the Secret-Split Hash method’s reduction efficiency is based upon the degrees of freedom due to the size of the hint.

![Diagram of Secret-Split Hash Method](image)

Figure 6.12: First step example: reconstruct the set of hint shares from the original hint, $H_1$ and this share’s hint share, $hs_{11}$. This allows the algorithm to pre-filter out those candidate shares from server 2 whose hint share $hs_{12}$ does not match the required value. Note: in this context, + refers to secret-split reconstruction.

The first step is to reconstruct the full set of hint shares for each share on server $i$ by combining the original hint with this share’s hint share, as shown in Figure 6.12. To identify the potential sibling shares on the next server to be processed, the appropriate hint share for that server is used to get the set of shares stored in that server’s pre-filtering hash table. For example, in order to find the pre-filtered set of shares on server 2, $hs_{12}$ can be used to perform a hash table lookup on server 2.
Figure 6.13: The seven test reconstructions performed when considering three candidate shares, assuming a hint splitting threshold of two. Note: in this context, + refers to secret-split reconstruction.
Once pre-filtering is complete, the list of candidate shares can be reduced further by performing a series of test reconstructions as shown in Figure 6.13. If any of these tests fail, they cannot be siblings. However, if they match, then these three shares might be siblings; the effects of false positives are discussed in more detail in Section 6.4.3.

In order to derive the general equation for the number of reconstruction attempts at any step we begin by looking at the simple example of test recombining two shares, share\(_1\) and share\(_2\) when the reconstruction threshold is two. The hint for share\(_1\) can be computed by combining the hint shares \(h_{s11}\) and \(h_{s12}\), which can be found on share\(_1\) and share\(_2\) respectively. Likewise, the hint for share\(_2\) can be computed by combining the hint shares \(h_{s21}\) and \(h_{s22}\). In this way, it can be seen that there are two possible reconstruction tests when testing whether two shares are candidate siblings.

To continue with this example, we now look at the number of recombination tests when testing three shares. The hint for share\(_2\) can be reconstructed using hint share \(h_{s22}\) two ways: with hint shares \(h_{s21}\) and \(h_{s23}\). Likewise, the hint for share\(_3\) can be reconstructed using hint share \(h_{s32}\) two ways: with hint shares \(h_{s31}\) and \(h_{s33}\). In this way, it can be seen that the number of these combinations can be generalized as two instances of two choose one, given a starting share. The remaining recombinations involve computing each of the three hints, \(H_1\), \(H_2\), and \(H_3\) using each of the corresponding hint shares, \([h_{s11}, h_{s12}, h_{s13}]\), \([h_{s21}, h_{s22}, h_{s23}]\), and \([h_{s31}, h_{s32}, h_{s33}]\) respectively. The number of these combinations can be generalized as three instances of two choose two, given a starting share. Figure 6.13 is a illustration of these recombination tests.

This pattern continues for subsequent sibling share test combinations as well as for higher thresholds. The only difference when using higher reconstruction thresholds is that the algorithm must bypass the tests afforded by lower thresholds. As a result, the general formula for the number of reconstruction attempts, \(a\), given \(x\) shares, with a reconstruction threshold of \(T\), is defined in the following equation:

\[
a = \sum_{i=T-1}^{x-1} \binom{x-1}{i} (i+1)
\]

The next step is to understand how the search space of potential candidate shares is reduced based on the given number of reconstruction tests. A single reconstruction test can be thought of as recombining two bit strings of length \(b\), with the goal of creating the target bit string of length \(b\). Since all outcomes are equally likely, the probability of obtaining the target bit string has probability, \(\rho(t)\), and is defined in Equation 6.9.
\[ \rho(t) = \left( \frac{1}{2^b} \right) \]  

(6.9)

As a result, the general formula for the space reduction, \( r \), is defined by Equation 6.10, where \( a \) is the number of reconstruction attempts and \( b \) is the size of the hint.

\[ r = \left( \frac{1}{2^b} \right)^a = 2^{-ab} \]  

(6.10)

The previous example illustrates the seven basic reconstruction tests performed for each set of shares, which would result in a space reduction of \( 2^{-7b} \). As more servers’ contents are taken into account, the combinations of hints that can be reconstructed increases. For example, with a splitting threshold of three, if five servers’ contents are needed during this method, the space reduction increases to \( 2^{-39b} \).
Figure 6.14: Reconstruction tests required for an 8 bit hint, $b = 8$, and varying share counts per server. Increasing the splitting threshold improves security at the cost of increasing the number of reconstruction tests that need to be conducted due to the larger reconstruction space.

This model was then tested by varying the size of the hint, $b$, for several hint splitting thresholds for a wide range of number of shares per server. Figure 6.14 shows the model for $b = 8$ and the resulting number of hint reconstruction tests that were required.

It can be seen that for $10^6$ shares per server, it will take roughly $10^{10}$ reconstruction tests when using a hint splitting threshold of two, but will require $10^{18}$ reconstruction tests for the same size server using a hint splitting threshold of four; this illustrates that increasing the hint splitting threshold increases security, because an additional server would need to compromised, but at the cost of performance. Furthermore, the jump in the number of reconstruction tests using a threshold of two occurs when there are sufficient to shares per server to cause the algorithm to require the contents of a third server since not all sibling shares have been identified yet. The jump is due to the large change in reconstruction space when considering candidate shares from the additional server. In this example, the jump occurs at $2^{24}$ shares per server, but in general always has the potential to occur at $2^{BS}$, where $b$ is the size of the hint and $S$ is the number of servers currently involved in
Figure 6.15: Example showing the state when the degrees of freedom afforded by the hint are greater than the number of shares per server. In this example, this occurs at $10^{9.6} \approx 2^{32}$.

Figure 6.15 illustrates the effect of increasing the hint size to the point that the degrees of freedom afforded by the hint size are larger than the number of shares per server. In this example, this occurs at $10^{9.6} \approx 2^{32}$. The implications of this phenomenon are discussed in Section 6.4.3, but in general server to illustrate the useful upper bound of the hint size in relation to the number of shares per server.

This method was then implemented using the JErasure [62] library to facilitate all secret-splitting operations via a Reed-Solomon [42,59–61,63,67] implementation using a Cauchy [30,41,65] matrix for the generator matrix; this was done to ensure the matrix and all of it sub-matrices are always invertible. The Secret-Split Hash method’s implementation was then validated against the theoretical model by running multiple performance tests, the results of which are depicted as the black line in Figure 6.16. It can be seen that the theoretical model accurately predicted how the implementation would perform.
Figure 6.16: The Secret-Split Hash method implemented and validated against the theoretical model. The black solid line denotes the mean performance with the sample population fitting a \( t \)-distribution.

6.4.3 Runtime Analysis

Figure 6.17 illustrates the Secret-Split Hash method’s primary benefit as compared to the previous method: a large performance increase at the cost of resistance to loose targeted theft, i.e. if loose targeted theft is not a concern, this method is able to run in linear time. It is worth noting that both methods can be tuned to perform at a wide array of both performance and security guarantees. In general, the design choice regarding hint size is based upon the trade-off between performance and resistance to loose targeted theft.

Algorithm 2 defines the process of how the Secret-Split Hash method reduces the search space required to identify sets of sibling shares. Line 2 sets up the initial candidate tuples such that each tuple consists of \( c \) hint shares, where \( c = T_h - 1 \), consisting of every combination of hint shares from \( c \) servers. Each of these tuples is tested against an incoming hint share from the next server in the datastore on line 8. In order to perform this test, the hints for each of the shares in the tuple,
Figure 6.17: Work required for varying hint sizes on a datastore with 1,000,000 shares per server. Performance can be maximized by choosing a hint size for the expected maximum size of the datastore, which will result in the optimal, i.e. linear, runtime. In this example, this occurs when $b = 20$.

as well as the incoming share, need to be reconstructed.

If all of the sibling tests pass, the incoming share is a possible sibling with the other shares in the tuple; as a result, it is added to the tuple as well as adding the tuple to the new candidates list for the following round. Once all shares on a given server are tested against each of the existing candidate tuples, the process continues to the next server, and continues until all sets of sibling shares have been identified.

6.5 Summary

These two novel disaster recovery methods strive to prevent data loss in a secret-split datastore where otherwise reconstruction might have been combinatorially prohibitive. Both the Set-Subset and Secret-Split Hash method were shown to outperform the current solution while providing higher availability, immunity to strict targeted theft, and providing customizable resistance to loose
targeted theft. Furthermore, the Set-Subset method was shown to have a relatively stable runtime over a wide range of resistance to loose targeted theft. In contrast, the Secret-Split Hash method’s runtime was shown to be very dependent upon the relationship between the hint size the number of files stored on each server, but was able to run in linear time if resistance to loose targeted theft was not a concern.
Algorithm 2 Secret-Split Hash Algorithm

1: function SecretSplitHash(st) \triangleright st: the hint splitting threshold
2: candidateTuples = INITIALIZE
3: step = 1
4: repeat
5: for each tuple in candidateTuples do
6: hintShares = server[st + step − 1]
7: for each hintShare in hintShares do
8: if SiblingTest(tuple, hintShare, st) then
9: tuple.push(hintShare)
10: newCandidates.push(tuple)
11: candidateTuples.clear()
12: candidateTuples = newCandidates
13: newCandidates.clear()
14: step = step + 1
15: until |candidateTuples| ≤ filesPerServer
16: function INITIALIZE
17: tempList
18: for each repoIndex do
19: for each hintShare do
20: if tempList.size() is hintShareIndex then
21: tempList.push(new Tuple(hintShare))
22: continue
23: tuple = server[repoIndex][hintShareIndex]
24: clone = tuple.clone()
25: clone.push(hintShare)
26: tempList.push(clone)
27: return tempList
28:
29: function SiblingTest(tuple, hintShare, st)
30: for st − 1 ≤ count ≤ |tuple| do
31: for choose count shares from tuple do
32: hint = RECONSTRUCT(shares, hintShare)
33: if hint ≠ origHint then
34: return false
35: return true
Chapter 7

Future Work and Conclusions

Despite the contributions made by this work, there are many opportunities to further their goals and improve upon their limitations. As such, we now present the top areas that we see as the next steps to continue this research, as well as the conclusions and lessons learned while conducting this work.

7.1 Future Work

There are several areas of research pertaining to evolutionary trends that were not covered by this work. First, we did not consider events due to system-directed operations, such as migration between system events [81]. Events of this type move information from one layer of the storage system to another (e.g., moving files from the disk cache to the primary tape silo). If these events are scheduled to occur without regard to user activity, i.e., uniformly across the work day or work week, then they will not tend to alter the shape of the overall workload curve, as Figure 4.4b shows. However, if these migration events are scheduled to occur during periods of expected low user activity, as is logical, then they will definitely mask the impact that user activity has on the system by further flattening out the overall workload on the system. In other words, the impact of users’ peak read times will be reduced, or even negated.

Locality of namespace accesses is another area of interest that is beyond the scope of this analysis. How are access statistics different when viewed in a namespace centric manner? For example, during a user’s access session, which files are accessed? Are they similar? Are they within a certain directory radius of each other?

Answers to these questions would better prepare system engineers to design the system to better meet the users’ needs. For example, if it is found that users most often access files within the
same or similar directories at the same time, then grouping files on media by directory or by user [40] and perhaps even pre-fetching them could dramatically improve read performance at relatively little cost, since the majority of the read cost is paid on access to the first byte.

Percival has several open areas of research. The first is to expand its support for additional access control systems. The current design only supports access control for situations in which there are a relatively low number of separate access credentials, i.e. less than 100. It would be ideal to expand the current design to fully support a user based access control system, one in which there may be several thousand access credentials. Furthermore, the current design does not support hierarchical access control. Reverse indexes are specific to the access credential under which they were ingested and/or updated. As a result, if multiple roles are authorized to perform a search for the same data object, that object’s identifier must be placed in the reverse index for each applicable access credential.

Another enhancement to Percival’s design is to support not only the presence of a keyword within a data object, but its locality as well. This will enhance Percival’s search capability beyond conjunctive searches by enabling exact phrase matching. It is theorized this can be accomplished by adding word locality information to each data object’s entry within a reverse index, thus allowing the client to test for adjacent locality in addition to taking a simply union of the available reverse indexes. The cost of such functionality is the increased size of each reverse index, which obviously impacts the space overhead of each query server, but also potentially increases a client’s reconstruction time due to larger indexes. The danger being that the increase in reconstruction time results in a loss of responsiveness during search operations; therefore extensive testing must be performed to ensure this does not occur.

Query server recovery time in the event a salt is compromised can also be improved. The time required to do so is not prohibitive, but it would require taking that query server offline for several days, which may impact data availability depending on the secret splitting scheme chosen.

While our performance testing shows that Percival is able to meet the needs of its user base in a timely fashion, it is by no means exhaustive. Testing Percival using a real world search workload and evaluating its performance would be an invaluable study.

The reconstruction methods proposed by this research can potentially be improved by parallelizing each of the methods. As shown previously, these methods are a large improvement over their predecessor, but they still have a relatively large processing time for large datastores, e.g. a datastore with $10^9$ shares per server requires approximately $10^{17}$ sibling tests using the Set-Subset method. As such, both would benefit by parallelizing the algorithms to run on Hadoop [36,95] or a similar system.
7.2 Conclusions

High-performance computing systems and storage systems have seen tremendous advances in computing power and storage capacity over the past two decades. Long-term storage is increasingly important in storing the results of research in such environments, yet no study has done an ‘apples-to-apples’ comparison of a single environment over such a long period of time.

This work began by comparing trace data for the NCAR center from 1992 to trace data taken from the current archive to determine evolutionary changes over the previous twenty years. The study produced several key findings that are relevant for designers building archival storage systems.

First, writes have become four times more frequent relative to reads over the past twenty years. This, combined with the reduction in the fraction of the archive that is actually accessed over three years, indicates that archives are becoming increasingly “write-only”, with attendant implications for system design.

In addition, the high level of writes suggests that systems should be designed to handle the high write load, with writes postponed during periods of high read activity. Since reads are primarily user-driven, these periods are highly predictable, and can allow system designers to save money by reducing the maximum level of concurrency the system must support.

Finally, as is well-documented, access latencies are not declining very fast. Given the bursty nature of reads, it may be useful to design systems to group or pre-fetch data to reduce perceived latency, even if doing so means reading data from archive to disk cache that may never be used. It is also necessary to use a relatively large disk cache to hide this latency from users, perhaps even permanently caching small files to reduce access latency for them. Fortunately, this approach is cost-effective given the relatively low cost of disk.

By studying the same storage system being used for the same purpose at two different periods separated by nearly two decades, we have provided valuable insight into long-term archival storage system behavior. We have also provided a detailed look at current user behavior for archival storage systems. In doing so, we hope to enable archival storage system designers to build long-term storage for the next twenty years and beyond.

Maintaining information privacy is difficult when sharing data across a distributed, long-term datastore operating in an untrusted environment. To address this need, we have presented Percival, a system that is designed to be applied to new or existing secret-split datastores, operates while compromised, minimizes insider threat, localizes data release upon compromise of an access control credential, and still provides accurate and timely search results; this is all achieved with a space overhead of a few terabytes per query server.
During the course of developing Percival, we made several discoveries, the first being that natural language processing can be quite nebulous as well as time consuming. This discovery reinforced our initial belief that pre-ingestion corpus processing should be left up to, and tailored for, the particular instance to which Percival is being applied.

Another lesson learned was that nothing is inviolate. Every aspect of a system must be assumed to be able to be compromised, regardless of the security guarantees associated with that subsystem. The natural extension of Percival’s management of query shares, specifically the distribution of information such that there is a high barrier to cross prior to information release, was to apply the same concept to its handling of the private aspects of the design, i.e. the salts. The breakthrough in Percival’s design came when we realized that is possible to keep even the salts isolated from each other; this design change converts full data release into a localized loss of privacy, and is what truly sets Percival apart from previous works.

We have presented two novel disaster recovery methods to prevent data loss in a secret-split datastore where otherwise reconstruction would have been combinatorially prohibitive. Both the Set-Subset and Secret-Split Hash method were shown to vastly outperform the current solution while providing higher availability and immunity to strict targeted theft. Furthermore, each method is relevant for usage in different situations, thereby providing system designers with customizable options when choosing a disaster recovery methodology.

While developing these methods to prevent data loss in secret-split datastore, several new discoveries were made. The first of which was that the original definition of targeted theft, which was defined simply as an attacker identifying, and subsequently accessing, small amounts of shares without detection, was both vague and insufficient. It did not properly address the attack vector in POTSHARDS, nor did it properly account for insider threat.

The second discovery that was made when implementing the Secret-Split Hash method was the large disparity in the average reconstruction rates between standard secret-splitting libraries. The first library we used in this work was the Crypto++ [92] library. However, due to its high runtime, we abandoned it and switched to using the JErasure [62] library, which is built upon GF-Complete [64], for all experiments since it was found to perform three orders of magnitude faster than the Crypto++ library. Both of these libraries are published, and used in production environments. However, the disparity in runtimes illustrates the cost of abstraction via Crypo++ over the low level JErasure library.

As society needs to store an ever-increasing volume of potentially sensitive data for a long time, we will need to find methods that can maintain data privacy and integrity in spite of security events; this will necessitate distributed approaches such as Percival. The techniques for secure
Figure 7.1: Comparison of the average reconstruction throughput between the Crypto++ and JErasure libraries during 1000 tests. In general, the JErasure library performed three orders of magnitude faster than the Crypto++ library.

searches developed for this work will help to make such datastores much more usable, ensuring that the long-term data in them will not be simply stored, but rather be available for effective access and use via search queries. By increasing the utility and value of long-term data storage, this approach can make it cost-effective to maintain secure long-term archives.
Bibliography


[50] Melville, H. *Moby-Dick*. Richard Bentley (Britain) and Harper and Brothers (US), 1851.


